WebSphere® Application Server for IBM i, Version 7.0

# Developing and deploying applications

**Note**

Before using this information, be sure to read the general information under “Notices” on page 469.

**Compilation date: September 16, 2008**

**© Copyright International Business Machines Corporation 2008.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

**How to send your comments** . . . vii

**Changes to serve you more quickly** . . . ix

**Chapter 1. Overview and new features for developing and deploying applications** . . . 1
Assembly tools . . . 1
Enterprise (Java EE) applications . . . 1
System applications . . . 2
Common deployment framework . . . 2
Business-level applications . . . 3
Assets . . . 5
EJB 3.0 deployment overview . . . 6
Service Data Objects: Resources for learning . . . 7

**Chapter 2. Debugging applications** . . . 9
Debugging components in the IBM Rational Application Developer for WebSphere . . . 10

**Chapter 3. Assembling applications** . . . 11
Application assembly and enterprise applications . . . 12
Assembly tools . . . 13
Generating code for Web service deployment . . . 13
Assembling applications: Resources for learning . . . 14

**Chapter 4. Class loading** . . . 17
Class loaders . . . 17
Configuring class loaders of a server . . . 21
Class loader collection . . . 23
Class loader ID . . . 23
Class loader order . . . 23
Class loader settings . . . 23
Configuring application class loaders . . . 24
Configuring Web module class loaders . . . 25
Class loading: Resources for learning . . . 27

**Chapter 5. Deploying and administering enterprise applications** . . . 29
Enterprise (Java EE) applications . . . 29
System applications . . . 30
Common deployment framework . . . 30
Installing enterprise application files . . . 31
Installable enterprise module versions . . . 32
Ways to install enterprise applications or modules . . . 33
Installing enterprise application files with the console . . . 36
Example: Installing an EAR file using the default bindings . . . 60
Example: Installing a Web Services Sample with the console . . . 61
Installing enterprise modules with JSR-88 . . . 63
Customizing modules using DConfigBeans . . . 64
Enterprise application collection . . . 65
Name . . . 66
Application Status . . . 66
Startup order . . . 67
Enterprise application settings . . . 67
Configuring enterprise application files . . . 68
Application bindings . . . 69

© Copyright IBM Corp. 2008

Configuring application startup. . . . 74
Configuring binary location and use . . . . 76
Configuring the use of class loaders by an application . . . . 81
Manage modules settings . . . . 85
Mapping modules to servers . . . . 87
Mapping virtual hosts for Web modules . . . . 88
Mapping properties for a custom login or trusted connection configuration. . . . . 90
Viewing deployment descriptors . . . . 91
Metadata for module settings . . . . 93
Starting or stopping enterprise applications . . . . 94
Disabling automatic starting of applications . . . . 95
Target specific application status . . . . 96
Exporting enterprise applications . . . . 98
Exporting enterprise application files . . . . 99
Exporting DDL files . . . . 100
Updating enterprise application files . . . . 100
Ways to update enterprise application files. . . . . 101
Updating enterprise applications with the console . . . . 103
Preparing for application update settings . . . . 105
Hot deployment and dynamic reloading . . . . 109
Uninstalling enterprise applications. . . . . 116
Removing enterprise files . . . . 117
Deploying and administering applications: Resources for learning . . . . 118

**Chapter 6. Managing applications through programming** . . . . 119
Application management . . . . 120
Accessing the application management function. . . . . 121
Installing an application through programming . . . . 122
Starting an application through programming . . . . 125
Uninstalling an application through programming . . . . 126
Manipulating additional attributes for a deployed application . . . . 129
Sharing sessions for application management . . . . 131
Updating an application through programming . . . . 132
Adding to, updating, or deleting part of an application through programming . . . . 134
Editing applications . . . . 136
Preparing a module and adding it to an existing application through programming . . . . 138
Preparing and updating a module through programming. . . . . 141
Deleting a module through programming . . . . 144
Adding a file through programming . . . . 146
Updating a file through programming . . . . 148
Deleting a file through programming . . . . 151
Extending application management operations through programming . . . . 153

**Chapter 7. Deploying and administering business-level applications** . . . . 157
Business-level applications . . . . 157
Assets . . . . 160
Composition units . . . . 160
Importing assets . . . . 161
Upload asset settings . . . . 163
Asset settings . . . . 164
Managing assets . . . . 167
Asset collection. . . . . 167
Updating assets . . . . 168
Deleting assets . . . . 171
Exporting assets . . . . 171
Creating business-level applications . . . . 172

Creating business-level applications with the console . . . . . . 173
Business-level application settings . . . . . . 182
Composition unit settings . . . . . . 184
Example: Creating a business-level application . . . . . . 185
Starting business-level applications . . . . . . 186
Stopping business-level applications . . . . . . 187
Updating business-level applications . . . . . . 187
Deleting business-level applications . . . . . . 189

**Chapter 8. Administering business-level applications using programming** . . . . . . 191
Creating an empty business-level application using programming . . . . . . 193
Importing an asset using programming . . . . . . 196
Adding a composition unit using programming . . . . . . 201
Starting a business-level application using programming. . . . . . . 208
Stopping a business-level application using programming . . . . . . 211
Checking the status of a business-level application using programming . . . . . . 214
Deleting a business-level application using programming . . . . . . 219
Deleting an asset using programming . . . . . . 222
Deleting a composition unit using programming . . . . . . 226
Exporting an asset using programming . . . . . . 230
Listing assets using programming . . . . . . 233
Listing composition units using programming . . . . . . 237
Listing business-level applications using programming . . . . . . 241
Editing a composition unit using programming . . . . . . 244
Editing an asset using programming . . . . . . 250
Editing a business-level application using programming . . . . . . 254
Updating an asset using programming . . . . . . 258
Viewing a composition unit using programming . . . . . . 262
Viewing an asset using programming. . . . . . . 266
Viewing a business-level application using programming. . . . . . . 269
Listing control operations using programming. . . . . . . 273

**Chapter 9. Troubleshooting deployment**. . . . . . . 279
Application deployment problems . . . . . . 279
Application deployment troubleshooting tips . . . . . . 283
A client program does not work . . . . . . 284
Application startup errors . . . . . . 285
Application startup problems . . . . . . 289
Web resource is not displayed . . . . . . 292
Application uninstallation problems. . . . . . . 294

**Chapter 10. Adding logging and tracing to your application** . . . . . . 295
Configuring Java logging using the administrative console . . . . . . 296
Java logging . . . . . . 296
Log level settings . . . . . . 297
Loggers . . . . . . 299
Log handlers. . . . . . . 300
Log levels. . . . . . . 300
Log filters . . . . . . 301
Log formatters . . . . . . 301
Using loggers in an application . . . . . . 302
HTTP error, FRCA, and NCSA access log settings . . . . . . 314
Logger.properties file for configuring logger settings . . . . . . 315
Example: Sample security policy for logging . . . . . . 316
Configuring applications to use Jakarta Commons Logging. . . . . . . 317
Jakarta Commons Logging . . . . . . 318

Configurations for the WebSphere Application Server logger . . . . . . . . . . . . . . . . 320
Programming with the JRas framework . . . . . . . . . . . . . . . . . . . . . . . . . . 323
JRas logging toolkit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 324
JRas Extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 326
JRas messages and trace event types . . . . . . . . . . . . . . . . . . . . . . . . . 334
Instrumenting an application with JRas extensions . . . . . . . . . . . . . . . . . . . . 336
Logging Common Base Events in WebSphere Application Server . . . . . . . . . . . . . . . . 343
The Common Base Event in WebSphere Application Server . . . . . . . . . . . . . . . . . 343
Logging with Common Base Event API and the Java logging API . . . . . . . . . . . . . . . 356
java.util.logging -- Java logging programming interface . . . . . . . . . . . . . . . . . 365
Logger.properties file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 366
Logging Common Base Events in WebSphere Application Server . . . . . . . . . . . . . . . 367

**Chapter 11. Securing Web services applications using the WSS APIs at the message level** 369
Securing messages at the request generator using WSS APIs . . . . . . . . . . . . . . . . 372
Configuring encryption to protect message confidentiality using the WSS APIs . . . . . . . . 372
Configuring generator signing information to protect message integrity using the WSS APIs . . . 387
Attaching the generator token using WSS APIs to protect message authenticity . . . . . . . . 405
Securing messages at the response consumer using WSS APIs . . . . . . . . . . . . . . . . 416
Configuring decryption to protect message confidentiality using the WSS APIs . . . . . . . . 417
Verifying consumer signing information to protect message integrity using WSS APIs . . . . . . 430
Validating the consumer token to protect message authenticity . . . . . . . . . . . . . . 448
Configuring Web services security using the WSS APIs . . . . . . . . . . . . . . . . . . 457
Web services security APIs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 459
Web services security configuration considerations when using the WSS API . . . . . . . . . 461
Encrypted SOAP headers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 462
Signature confirmation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 464

**Appendix. Directory conventions** . . . . . . . . . . . . . . . . . . . . . . . . . 467

**Notices** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 469

**Trademarks and service marks** . . . . . . . . . . . . . . . . . . . . . . . . . . 471

# How to send your comments

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

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

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

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

© Copyright IBM Corp. 2008

# Changes to serve you more quickly

**Print sections directly from the information center navigation**

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

To print a section of the information center navigation:

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

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

**Under construction!**

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

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

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

© Copyright IBM Corp. 2008

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

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

**What is new for developers**

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

**Learn about WebSphere applications: Overview and new features**

This topic provides an overview of the programming model.

**Accessing the Samples (Samples Gallery)**

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

## Assembly tools

WebSphere® Application Server supports *assembly tools* that you can use to develop, assemble, and deploy Java™ Platform, Enterprise Edition (Java EE) modules.

The IBM® Rational® Application Developer for WebSphere Software product provides supported assembly tools.

Although this information center refers to Rational developer products as the *assembly tools*, you can use the products to do more than assemble modules. Rational Application Developer is an integrated development environment that provides development, testing, assembly and deployment capabilities. Rational Application Developer provides extensive online documentation. Topics on application assembly in this information center supplement that documentation, focusing on assembling Java EE modules using the Java EE Perspective of the assembly tools.

Rational Application Developer is available in the WebSphere Application Server disc package with two licenses. The license for assembly and deployment capabilities does not expire. The license for development and other capabilities is available on a Trial basis and is only available for a limited time.

The Trial download for Rational Application Developer is available at http://www.ibm.com/developerworks/downloads/r/rad/.

**Note:** The assembly tools run on Windows® and Linux® Intel® platforms. Users of WebSphere Application Server on all platforms must assemble their modules using an assembly tool installed on Windows or Linux Intel platforms. To install an assembly tool, follow instructions available with the tool.

## Enterprise (Java EE) applications

Enterprise applications (or Java EE applications) are applications that conform to the Java Platform, Enterprise Edition (Java EE) specification. Prior to Java EE 5, the specification name was Java 2 Platform, Enterprise Edition (J2EE). The term *Java EE* includes Java EE 5 and J2EE specifications.

Enterprise applications can consist of the following:
- Zero or more EJB modules (packaged in JAR files)
- Zero or more Web modules (packaged in WAR files)
- Zero or more connector modules (packaged in RAR files)
- Zero or more Session Initiation Protocol (SIP) modules (packaged in SAR files)
- Zero or more application client modules

© IBM Corporation 2003

- Additional JAR files containing dependent classes or other components required by the application
- Any combination of the above

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

## System applications

A *system application* is a Java Platform, Enterprise Edition (Java EE) enterprise application that is central to a WebSphere Application Server product.

Examples of system applications include *isclite*, *managementEJB* and *filetransfer*.

Because a system application is an important part of a WebSphere Application Server product, a system application is deployed when the product is installed and is updated only through a product fix or upgrade. For some system applications, such as *filetransfer*, users cannot change the metadata for the system application, unless the metadata assigns users and groups for security purposes. For these applications, non-security related metadata such as its Java EE bindings or extensions must be updated through a product fix or upgrade.

System applications are not shown in the list of installed applications on the console Enterprise Applications page, or through wsadmin and Java application programming interfaces, to prevent users from accidentally stopping, updating or removing the system applications.

Note that Java EE Samples are not system applications even though they are provided as part of a WebSphere Application Server product. Similarly, applications that support changes to their metadata are not system applications.

## Common deployment framework

The *common deployment framework* enables you to implement plug-ins that add steps to default Java Platform, Enterprise Edition (Java EE) application management operations such as install, uninstall, edit and update.

Using the framework, you can implement management operations on specific types of deployable contents. For example, the deployable contents might include EAR, WAR, JAR or other Java EE modules and the management operations might include install and uninstall. Each operation is divided into a number of steps. For example, the install operation has steps for EJBDeploy and JavaServer Pages (JSP) compilation, among others. Using the common deployment framework, you can add steps to the default logic for Java EE operations.

The product supports framework plug-ins that extend deployment of EAR files. An EAR file has operations such as createEarWrapper, installApplication, uninstallApplication and editApplication. Using a framework plug-in, you can add steps to default install operations that support, for example, creating additional configuration artifacts in a configuration session, modifying an input EAR file using code generation, or additional validating of input parameters.

To extend application management operations using the framework, a plug-in must do the following:

- Implement each step.

  A *step* runs logic that performs an operation. A step can access the deployment context and the deployable object. The *deployment context* provides information such as the operation name, the configuration session identifier, the temporary location for creating temporary files, operations parameters, and the like. A step is added by the extension provider.
- Implement an extension provider that adds each implemented step.

  An *extension provider* is a class that provides steps for an operation on a given type, the EAR file type.
- Register the plug-in with a WebSphere Application Server server.

The plug-in is implemented as an Eclipse plug-in and is placed in *app_server_root*/`plugins` directory. Add the extension point for the extension provider in the `META-INF/plugin.xml` file within the plug-in JAR file.

For an example of these steps, refer to "Extending application management operations through programming" on page 153.

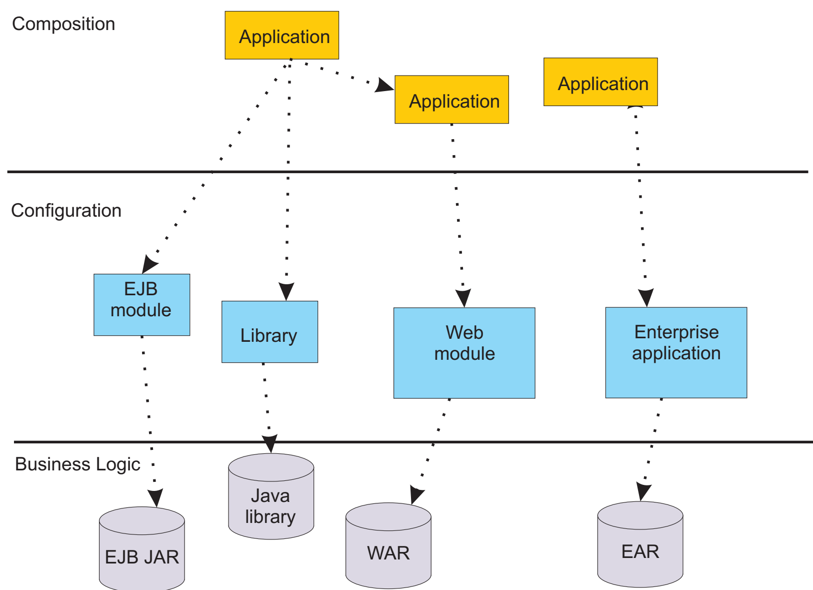
## Business-level applications

A business-level application is an administration model that provides the entire definition of an application as it makes sense to the business. A business-level application is a WebSphere configuration artifact, similar to a server or cluster, that is stored in the product configuration repository.
- Business-level application characteristics
- Comparisons to Java EE applications

### Business-level application characteristics

A business-level application has the following characteristics:
- A business-level application is an administration model of the definition of an enterprise-level application that consists of WebSphere and non-WebSphere artifacts. The business-level application might not explicitly manage the lifecycle of every artifact. It is a model for defining an application.
- A business-level application does not represent or contain application binary files. It is a configuration that lists one or more composition units, which represent the application binary files. A business-level application uses the binary files to run the application business logic. Administration of binary files is separate from administration of the application definition.
- A business-level application supports recursive composition by reference that facilitates hierarchical assembly of business-level applications and individual deployed artifacts within or outside a WebSphere product. The composition at its lowest level consists of configured instances of application binary files that run in a specific runtime environment such as an application server. Installable packages or archives, such as Java archives (JAR) or enterprise archive (EAR) files, typically deliver the business logic that these configured instances represent to corresponding runtime platforms.

The following diagram shows the composition model for business-level applications:




A business-level application does not introduce new programming, runtime, or packaging models:

- You do not need to change your application business logic. The business-level application function does not introduce new application programming interfaces (APIs).
- You do not need to change your application runtime settings. The product supports all of the runtime characteristics, such as security, class loading and isolation, required by individual programming models to which business components are written.
- You do not need to change your application packaging. There is no specific unique packaging model that provides a business-level application definition.

Typically, you first create an empty business-level application and then add composition units to it. The business-level application name must be unique within a cell. The business level application itself has minimal configuration data associated with it, solely the list of composition units, but individual composition units might save application-specific configuration data.

A business-level application is defined in the product configuration repository under *app_server_root*/cells/ *cell_name*/blas/*business_level_application_name*/bver/BASE/bla.xml.

## Comparisons to Java EE applications

Business-level applications can consist of or aggregate Java Platform, Enterprise Edition (Java EE) applications and modules with non-Java EE artifacts. The contents of Java EE applications integrate with business-level application concepts for deployment and management of applications. Existing Java EE application management APIs continue to work after you add Java EE application or modules to a business-level application. The business-level application management API accepts Java EE contents and configurations and delegates to existing Java EE management APIs. Control operations such as starting and stopping a Java EE composition unit are delegated to ApplicationManager MBean on application servers that start and stop Java EE applications.

*Table 1. Java EE concepts compared to business-level application concepts*

| Java EE concept | Business-level application concept | Description |
| --- | --- | --- |
| EAR or stand-alone module for deployment | Asset | Java EE application contents are assets. |
| Java EE application created at the end of application install | Composition unit | A Java EE application is in an enterprise archive (EAR) file. The product saves the EAR file in the product repository as a composition unit. |
| Java EE modules within the EAR file | Deployable units in the asset | Each module in the EAR file is a deployable unit that you can install on independent deployment targets. The EAR file is still managed as a single asset in its entirety. |

*Table 1. Java EE concepts compared to business-level application concepts (continued)*

| Java EE concept | Business-level application concept | Description |
| --- | --- | --- |
| Java EE application installation using the administrative console, programming, or wsadmin commands | Multiple business-level application management commands<br><br>During Java EE application deployment, you can specify the name of the business-level application to include the Java EE application. If the business-level application name is not set, the product creates a default business-level application with the same name as the Java EE application name. The product adds a composition unit with the same name as the Java EE application name under the business-level application. You can deploy multiple Java EE applications under a single business-level application. | You can make a Java EE application a business-level application and add it to another business-level application:<br>1. Install the Java EE application (EAR file) using the enterprise application installation console wizard, programming, or wsadmin. Keep the default selection to create a business-level application that has the same name as the Java EE application.<br>2. Create an empty business-level application.<br>3. Add the EAR file business-level application to the empty business-level application. The EAR file business-level application is a composition unit of the containing business-level application.<br><br>Or, you can make a Java EE application an asset and add it to another business-level application:<br>1. Import an EAR file as an asset. It has an asset type aspect of `Java EE ear`.<br>2. Create an empty business-level application.<br>3. Add the Java EE application asset to the business-level application. The EAR file asset is a composition unit of the containing business-level application.<br>4. Collect targets for each deployable unit (Java EE module). |
| Uninstall Java EE application | Multiple business-level application management commands | You delete the Java EE application composition unit from the business-level application:<br>1. Remove the composition unit for the Java EE application from the business-level application.<br>2. If the EAR file is an asset, delete the asset. |
| Start the Java EE application. | Start the composition unit. | Starting a business-level application starts any Java EE application in it. |
| Stop the Java EE application. | Stop the composition unit. | Stopping a business-level application stops any Java EE application in it. |

## Assets

An asset represents one or more application binary files that are stored in an asset repository. Typical assets include application business logic such as Java Platform, Enterprise Edition (Java EE) archives, library files, and other resource files.

An asset repository stores the binary files for the asset. The product configuration repository provides a default asset repository.

Assets in the configuration repository are managed by the product management domain. The configuration repository stores asset binary files in *app_server_root*/config/cells/*cell_name*/assets/*asset_name*/aver/*BASE*/bin/.

An asset name must be unique within a cell, the product administrative domain.

The product creates an asset.xml file when an asset is registered with the product configuration. The file contains information about the asset such as its name, destination location, and dependencies on other assets.

You must register files as assets before you can add them to one or more business-level applications. At the time of asset registration, you can import the physical application files into the product configuration repository or you can specify an external location where the asset resides.

## EJB 3.0 deployment overview

Learn about the Enterprise JavaBeans (EJB) 3.0 deployment model, including to *Just-In-Time* (JIT) deployment.

All Java Platform, Enterprise Edition (Java EE ) application server products have some form of EJB deployment phase where your application is customized to run in that particular application server implementation. Typically, this is accomplished by an application server-specific deployment tool, which generates code to bridge your EJB interface and implementation code to the application server's EJB container implementation. Some application server products' deploy tools alter the bytecodes of your application classes rather than using code generation, but the end result is similar.

In WebSphere Application Server, the bridging is accomplished by generating code that *wraps* your EJB implementation classes, connecting them to the product EJB container, which in turn allows the EJB container to host your enterprise beans and provide services to them. If one or more of your enterprise beans has defined remote interfaces, additional code is generated to provide the remote function.

Historically, EJB deployment in the WebSphere product has been performed by the *EJBDeploy* tool that is included with product and packaged with WebSphere product-oriented development tools.

The EJBDeploy tool introspects your EJB external interfaces, generates the wrapper code as .java files, then compiles it using the javac compiler to produce .class files, which are then packaged in your EJB module with your application code. The EJBDeploy tool also runs the rmic tool against the remote EJB interfaces in the application, producing additional *stub* and *tie* class files that interact with the Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) Object Request Broker (ORB), providing remote object support. Typically, you run the EJBDeploy tool either when you install the application on the product or sometime before you install the application from the command-line tool or within a development tool.

**Just-In-Time deployment**

The EJB 3.0 support in WebSphere Application Server adds a new feature called Just-In-Time Deployment. With Just-In-Time Deployment, the EJB container dynamically generates the wrapper, stub, and tie classes in-memory as needed when the application is running. Additionally, the Web container and application client containers dynamically generate the stub class required for remote EJB invocations. Effectively, this means that you do not need to process EJB 3.0 modules, Web modules that invoke EJB 3.0 beans, or client modules that invoke EJB 3.0 beans, through the EJBDeploy tool prior to running them in WebSphere.

**createEJBStubs tool**

Even though the Just-In-Time Deployment feature, in many cases, dynamically generates the RMI-IIOP stub classes that are required for invocation of remote EJB interfaces, there remain some cases where these stub classes are not dynamically generated. For EJB 3.0 clients not running inside a Web container, EJB container, or client container, that is upgraded to EJB 3.0 level, you must generate the stub classes with the createEJBStubs tool, then make the generated stubs available in the client environment's classpath. Typically you would accomplish this by copying the generated stubs to the location where the client's business interface class resides.

To summarize, the createEJBStubs tool must be used to generate client-side stubs for the following environments:

- ″Bare″ Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.
- WebSphere Application Server container environments prior to Version 7 that do not have the Feature Pack for EJB 3.0 applied.
- Non-WebSphere Application Server environments.

For more information about packaging your EJB module, see the topic, ″EJB 3.0 module packaging overview.″

## Service Data Objects: Resources for learning

Use the following links to find relevant supplemental information about the service data object and various other functions that can be used with it. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

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

### Service Data Objects

For an introduction to Service Data Objects, refer to:
- Introduction to Service Data Objects

For an overview of the Service Data Objects specification, refer to:
- Service Data Objects

A good place to start to learn about the Eclipse Modeling Framework is:
- EMF Eclipse Modeling Framework

Information about XSD to SDO/EMF mapping for Version 6 can be found at:
- XML Schema to Ecore Mapping

### Web application presentation layer technologies

For a brief overview of JavaServer Faces, refer to:
- Java Sun J2EE 1.4 tutorial

Good places to start to learn about JavaServer Pages Standard Tag Library are:
- JavaServer Pages Standard Tag Library
- A JSTL primer, Part 1: The expression language

# Chapter 2. Debugging applications

To debug your application, you must use a development environment like the IBM Rational Application Developer for WebSphere to create a Java project. You must then import the program that you want to debug into the project.

## About this task

By following the steps below, you can import the WebSphere Application Server examples into a Java project. Two debugging styles are available:

- **Step-by-step** debugging mode prompts you whenever the server calls a method on a Web object. A dialog lets you step into the method or skip it. In the dialog, you can turn off step-by-step mode when you are finished using it.
- **Breakpoints** debugging mode lets you debug specific parts of programs. Add breakpoints to the part of the code that you must debug and run the program until one of the breakpoints is encountered.

Breakpoints actually work with both styles of debugging. Step-by-step mode just lets you see which Web objects are being called without having to set up breakpoints ahead of time.

You do not need to import an entire program into your project. However, if you do not import all of your program into the project, some of the source might not compile. You can still debug the project. Most features of the debugger work, including breakpoints, stepping, and viewing and modifying variables. You must import any source that you want to set breakpoints in.

The inspect and display features in the source view do not work if the source has build errors. These features let you select an expression in the source view and evaluate it.

1. Create a Java Project by opening the New Project dialog.
2. Select **Java** from the left side of the dialog and **Java Project** in the right side of the dialog.
3. Click **Next** and specify a name for the project, for example, WASExamples.
4. Click **Finish** to create the project.
5. Select the new project, choose **File > Import > File System**, then **Next** to open the import file system dialog.
6. Browse the directory for files.

   Go to the following directory: *profile_root*/installedApps/*node_name*/DefaultApplication.ear/DefaultWebApplication.war.
7. Select DefaultWebApplication.war in the left side of the Import dialog and then click **Finish**. This imports the JavaServer Pages files and Java source for the examples into your project.
8. Add any JAR files needed to build to the Java Build Path.

   Select **Properties** from the right-click menu. Choose the Java Build Path node and then select the Libraries tab. Click **Add External JARs** to add the following JAR files:
   - *profile_root*/installedApps/*node_name*/DefaultApplication.ear/Increment.jar.

     When you have added this JAR file, select it and use the **Attach Source** function to attach the Increment.jar file because it contains both the source and class files.
   - *app_server_root*/lib/j2ee.jar
   - *app_server_root*/plugins//com.ibm.ws.runtime.jar
   - *app_server_root*/plugins/com.ibm.ws.webcontainer.jar

   Click **OK** when you have added all of the JARs.
9. You can set some breakpoints in the source at this time if you like, however, it is not necessary as step-by-step mode will prompt you whenever the server calls a method on a Web object. Step-by-step mode is explained in more detail below.

© Copyright IBM Corp. 2008

10. To start debugging, you need to start the WebSphere Application Server in debug mode and make note of the JVM debug port. The default value of the JVM debug port is 7777.
11. When the server is started, switch to the debug perspective by selecting **Window > Open Perspective > Debug**. You can also enable the debug launch in the Java Perspective by choosing **Window > Customize Perspective** and selecting the **Debug** and **Launch** checkboxes in the **Other** category.
12. Select the workbench toolbar **Debug** pushbutton and then select **WebSphere Application Server Debug** from the list of launch configurations. Click the **New** pushbutton to create a new configuration.
13. Give your configuration a name and select the project to debug (your new WASExamples project). Change the port number if you did not start the server on the default port (7777).
14. Click **Debug** to start debugging.
15. Load one of the examples in your browser. For example: `http://`*`your.server.name`*`:`*`9080`*`/hitcount`

## What to do next

To learn more about debugging, launch the The IBM Rational Application Developer for WebSphere, select **Help > Help Contents** and choose the **Debugger Guide bookshelf** entry. To learn about known limitations and problems that are associated with the IBM Rational Application Developer for WebSphere, see the IBM Rational Application Developer for WebSphere release notes. For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

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

# Debugging components in the IBM Rational Application Developer for WebSphere

The IBM Rational Application Developer for WebSphere, included with the WebSphere Application Server on a separately-installable CD, includes debugging functionality that is built on the Eclipse workbench. Documentation for the IBM Rational Application Developer for WebSphere is provided with that product. To learn more about the debug components, launch the IBM Rational Application Developer for WebSphere, select **Help > Help Contents** and choose the **Developing > Debugging applications** bookshelf entries.

The IBM Rational Application Developer for WebSpheret includes the following:

**The WebSphere Application Server debug adapter**
: which allows you to debug Web objects that are running on WebSphere Application Server and that you have launched in a browser. These objects include enterprise beans, JavaServer Pages files, and servlets.

**The JavaScript™ debug adapter**
: which enables server-side JavaScript debugging.

**The Compiled language debugger**
: which allows you to detect and diagnose errors in compiled-language applications.

**The Java development tools (JDT) debugger**
: which allows you to debug Java code.

All of the debug components in the IBM Rational Application Developer for WebSphere can be used for debugging locally and for remote debugging. To learn more about the debug components, launch the IBM Rational Application Developer for WebSphere, select **Help > Help Contents** and choose the **Developing > Debugging applications** bookshelf entries.

# Chapter 3. Assembling applications

Application assembly consists of creating Java Platform, Enterprise Edition (Java EE) modules that can be deployed onto application servers. The modules are created from code artifacts such as Web application archives (WAR files), resource adapter archives (RAR files), enterprise bean (EJB) JAR files, and application client archives (JAR files). This packaging and configuring of code artifacts into enterprise application modules (EAR files) or stand-alone Web modules is necessary for deploying the modules onto an application server.

## Before you begin

This topic assumes that you have developed code artifacts that you want to deploy onto an application server and have unit tested the code artifacts in your favorite integrated development environment. Code artifacts that you might assemble into deployable Java EE modules include the following:
- Enterprise beans
- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (*connector*) implementations
- Application clients
- Session Initiation Protocol (SIP) modules (SAR files)
- Other supporting classes and files

To assemble your code artifacts into deployable Java EE modules, you can use a supported assembly tool. The product supports IBM Rational Application Developer for WebSphere Software for developing, assembling, and deploying Java EE modules.

## About this task

You assemble code artifacts into Java EE modules in order to deploy the code artifacts onto an application server. When you assemble code artifacts, you package and configure the code artifacts into deployable Java EE applications and modules, edit annotations or deployment descriptors, and map databases as needed. Unless you assemble your code artifacts into Java EE modules, you cannot run them successfully on an application server.

This topic describes how to assemble Java EE code artifacts into deployable modules using an assembly tool. Alternatively, you can use a rapid deployment tool to quickly assemble and deploy Java 2 Platform, Enterprise Edition (J2EE) 1.3 or 1.4 code artifacts. Refer to ″Rapid deployment of J2EE applications″ for details.

1. Start an assembly tool.
2. Optional: Read the online documentation for the assembly tool.
3. Configure the assembly tool for work on Java EE modules.
4. Migrate J2EE 1.4 or earlier projects or code artifacts created with the Application Server Toolkit, Assembly Toolkit, Application Assembly Tool (AAT) or a different tool.

   To migrate files, use the Migration wizard or import the files to the assembly tool.
5. Create an enterprise application project to which you can add archive files. You can create an enterprise application project separately or when you create archive files such as the following:
   - Create a Web project.
   - Create an enterprise bean (EJB) project.
   - Create an application client.
   - Create a resource adapter (connector) project.
6. Edit the annotations or deployment descriptors as needed. You can edit annotations or deployment descriptors for enterprise application, Web, application client, and enterprise bean (EJB) modules.

© IBM Corporation 2002 

Topics in Rational Application Developer documentation provide extensive information on editing annotations or deployment descriptors.

7. Optional: Generate enterprise bean (EJB) to relational database (RDB) mappings for EJB 2.1 or earlier modules.
8. Verify the archive files.
9. Generate code for deployment for Web services-enabled modules or for enterprise applications that use Web service modules.

### What to do next

After assembling your applications, use a systems management tool to deploy the EAR or WAR files onto the application server. "Ways to install enterprise applications or modules" on page 33 lists systems management tools available for deploying Java EE modules on an application server. The systems management tool follows the security and deployment instructions defined in the annotations or deployment descriptors, and enables you to modify bindings specified within an assembly tool. The tool locates the required external resources that the application uses, such as enterprise beans and databases.

Package your application so that the EAR file contains necessary modules only. Modules can include metadata for the modules such as information on annotations, deployment descriptors, bindings, and IBM extensions.

Use the administrative console at installation to complete the security instructions defined in the annotations or deployment descriptors and to locate required external resources, such as enterprise beans and databases. You can add configuration properties and redefine binding properties defined in an assembly tool.

## Application assembly and enterprise applications

Application assembly is the process of creating an enterprise archive (EAR) file containing all files related to an application. This configuration and packaging prepares the application for deployment onto an application server.

EAR files are comprised of the following archives:

- Enterprise bean JAR files (known as EJB modules)
- Web archive (WAR) files (known as Web modules)
- Application client JAR files (known as client modules)
- Resource adapter archive (RAR) files (known as resource adapter modules)
- SAR files (known as Session Initiation Protocol (SIP) modules)

Ensure that modules are contained in an EAR file so that they can be deployed onto the server. The exceptions are WAR modules, which you can deploy individually. Although WAR modules can contain regular Java archive (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 an annotation or deployment descriptor containing instructions for module deployment on the application server.

  You can use the graphical interface of Rational Application Developer assembly tools to generate the annotation or deployment descriptor. You can also edit annotations or descriptors directly in your favorite XML editor.
- Packaging modules into a single EAR file, which contains one or more files in a compressed format.

As part of the assembly process, you might also set environment-specific binding information. These bindings are defaults for an administrator to use when installing the application through the administrative

console. Further, you might define IBM extensions to the Java Platform, Enterprise Edition (Java EE) specifications, such as to allow servlets to be served by class name. To ensure portability to other application servers, these extensions are saved in an XML file that is separate from the standard annotation or deployment descriptor.

## Assembly tools

WebSphere Application Server supports *assembly tools* that you can use to develop, assemble, and deploy Java Platform, Enterprise Edition (Java EE) modules.

The IBM Rational Application Developer for WebSphere Software product provides supported assembly tools.

Although this information center refers to Rational developer products as the *assembly tools*, you can use the products to do more than assemble modules. Rational Application Developer is an integrated development environment that provides development, testing, assembly and deployment capabilities. Rational Application Developer provides extensive online documentation. Topics on application assembly in this information center supplement that documentation, focusing on assembling Java EE modules using the Java EE Perspective of the assembly tools.

Rational Application Developer is available in the WebSphere Application Server disc package with two licenses. The license for assembly and deployment capabilities does not expire. The license for development and other capabilities is available on a Trial basis and is only available for a limited time.

The Trial download for Rational Application Developer is available at http://www.ibm.com/developerworks/downloads/r/rad/.

**Note:** The assembly tools run on Windows and Linux Intel platforms. Users of WebSphere Application Server on all platforms must assemble their modules using an assembly tool installed on Windows or Linux Intel platforms. To install an assembly tool, follow instructions available with the tool.

## Generating code for Web service deployment

Before deploying Web services-enabled modules or any enterprise application archive (EAR) files that contain Web services-enabled module onto an application server, you must generate deployment code for the application.

### Before you begin

This article assumes you have assembled a module enabled with Web services, added it to an application, saved the application, and verified the application. It also assumes that you have started and configured an assembly tool.

### About this task

You can use an assembly tool to generate deployment code for the Web services-enabled module or for the EAR file that contains the Web services-enabled module.

1. If you have turned automatic validation off, manually validate any modules that use Web services with the JSR109 Web services validator before generating deployment code for them. If validating your module results in compilation errors or validation errors, fix the errors before generating deployment code. However, if validating your module results in warning or information messages, you can generate deployment code.
2. In the Project Explorer view of the assembly tool, right-click on the Web services-enabled module (WAR, enterprise bean JAR, or application client JAR file) for which you want to generate code for deployment.

3. Click **Deploy**. Alternatively, you can generate deployment code for Web services-enabled modules using the deployment tool for Web services (`wsdeploy`) from a command prompt.
4. If messages indicate that automatic file overwriting is not enabled, click **Yes to All** so the generated files are added to the module.
5. If errors such as *Unbound classpath variable: WAS_50_PLUGINDIR* display in the Tasks list, change the Java build path libraries properties to define that variable to be the WebSphere Application Server installation directory.

### Results

Code is generated into the folder where your Web services-enable module is located. Problems with the generation of code result in a window that displays error messages.

### What to do next

Install the Java Platform, Enterprise Edition (Java EE) application on your server machine. You can install the application onto a server using the administrative console. Before installing the application, you might need to set class paths.

## Assembling applications: Resources for learning

Additional information and guidance on assembling applications is available on various Internet sites.

Use the following links to find relevant supplemental information about the application assembly and using an assembly tool. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

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

View links to additional information about:
- "Programming instructions and examples"
- "Programming specifications"
- "Administration" on page 15

### Programming instructions and examples

- Rational Application Developer V7 Programming Guide, SG24-7501-00, http://www.redbooks.ibm.com/abstracts/sg247501.html?Open
- Rational developer community, http://www.ibm.com/developerworks/rational/
- *IBM WebSphere Developer Technical Journal:* Using Rational Developer to create a simple Web service and use it in a Web application, http://www.ibm.com/developerworks/websphere/techjournal/0506_parkin/0506_parkin.html
- Java EE Tutorials, http://java.sun.com/javaee/reference/tutorials/
- Recommended reading list: J2EE and WebSphere Application Server, http://www.ibm.com/developerworks/websphere/library/techarticles/0305_issw/recommendedreading.html
- Automated Deployment of Enterprise Application Updates: Part 1 - Basic concepts, http://websphere.sys-con.com/read/47889.htm

### Programming specifications

- Specifications and API documentation

## Administration

- *IBM WebSphere Developer Technical Journal:* System management for WebSphere Application Server V6 -- Part 1 Overview of system management enhancements, http://www.ibm.com/developerworks/websphere/techjournal/0501_williamson/0501_williamson.html
- *IBM WebSphere Developer Technical Journal:* System management for WebSphere Application Server V6 -- Part 5: Flexible options for updating deployed applications, http://www.ibm.com/developerworks/websphere/techjournal/0510_apte/0510_apte.html
- *WebSphere Application Server V6.1:* System Management Configuration Handbook, SG24-7304-00, http://www.redbooks.ibm.com/abstracts/SG247304.html?Open

# Chapter 4. Class loading

Class loaders are part of the Java virtual machine (JVM) code and are responsible for finding and loading class files. Class loaders enable applications that are deployed on servers to access repositories of available classes and resources. Application developers and deployers must consider the location of class and resource files, and the class loaders used to access those files, to make the files available to deployed applications. Class loaders affect the packaging of applications and the runtime behavior of packaged applications of deployed applications.

## Before you begin

This topic describes how to configure class loaders for application files or modules that are installed on an application server.

To better understand class loaders in WebSphere Application Server, read "Class loaders." The topic "Class loading: Resources for learning" on page 27 refers to additional sources.

## About this task

Configure class loaders for application files or modules that are installed on an application server using the administrative console. You configure class loaders to ensure that deployed application files and modules can access the classes and resources that they need to run successfully.

1. If an installed application module uses a resource, create a resource provider that specifies the directory name of the resource drivers.

   Do not specify the resource Java archive (JAR) file names. All JAR files in the specified directory are added into the class path of the WebSphere Application Server extensions class loader. If a resource driver requires a native library (`.dll` or `.so` file), specify the name of the directory that contains the library in the native path of the resource configuration.
2. Specify class-loader values for an application server.
3. Specify class-loader values for an installed enterprise application.
4. Specify the class-loader mode for an installed Web module.
5. If your deployed application uses shared library files, associate the shared library files with your application. Use a library reference to associate a shared library file with your application.
   a. If you have not done so already, define shared libraries for library files that your applications need.
   b. Define a library reference for each shared library that your application uses.

## What to do next

After configuring class loaders, ensure that your application performs as desired. To diagnose and fix problems with class loaders, refer to Troubleshooting class loaders.

# Class loaders

Class loaders find and load class files. Class loaders enable applications that are deployed on servers to access repositories of available classes and resources. Application developers and deployers must consider the location of class and resource files, and the class loaders used to access those files, to make the files available to deployed applications.

This topic provides the following information about class loaders in WebSphere Application Server:

- "Class loaders used and the order of use" on page 18
- "Class-loader isolation policies" on page 19
- "Class-loader modes" on page 21

© Copyright IBM Corp. 2008 

## Class loaders used and the order of use

The product runtime environment uses the following class loaders to find and load new classes for an application in the following order:

1. The bootstrap, extensions, and CLASSPATH class loaders created by the Java virtual machine

   The bootstrap class loader uses the boot class path (typically classes in jre/lib) to find and load classes. The extensions class loader uses the system property java.ext.dirs (typically jre/lib/ext) to find and load classes. The CLASSPATH class loader uses the CLASSPATH environment variable to find and load classes.

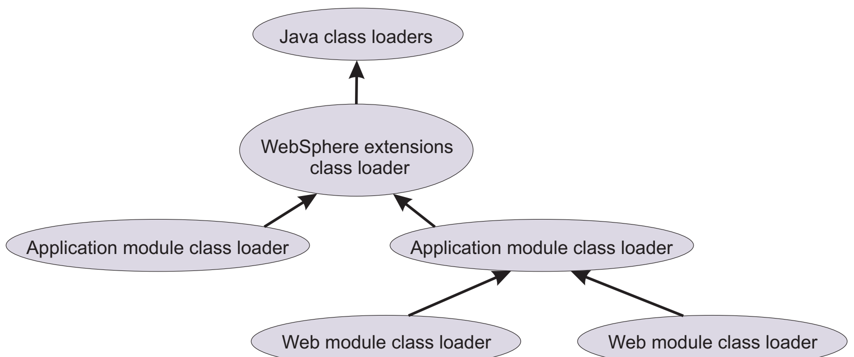
   The CLASSPATH class loader loads the Java Platform, Enterprise Edition (Java EE) application programming interfaces (APIs) provided by the WebSphere Application Server product in the j2ee.jar file. Because this class loader loads the Java EE APIs, you can add libraries that depend on the Java EE APIs to the class path system property to extend a server class path. However, a preferred method of extending a server class path is to add a shared library.
2. A WebSphere extensions class loader

   The WebSphere extensions class loader loads the WebSphere Application Server classes that are required at run time. The extensions class loader uses a ws.ext.dirs system property to determine the path that is used to load classes. Each directory in the ws.ext.dirs class path and every Java archive (JAR) file or ZIP file in these directories is added to the class path used by this class loader.

   The WebSphere extensions class loader also loads resource provider classes into a server if an application module installed on the server refers to a resource that is associated with the provider and if the provider specifies the directory name of the resource drivers.
3. One or more application module class loaders that load elements of enterprise applications running in the server

   The application elements can be Web modules, enterprise bean (EJB) modules, resource adapter archives (RAR files), and dependency JAR files. Application class loaders follow Java EE class-loading rules to load classes and JAR files from an enterprise application. The product enables you to associate shared libraries with an application.
4. Zero or more Web module class loaders

   By default, Web module class loaders load the contents of the WEB-INF/classes and WEB-INF/lib directories. Web module class loaders are children of application class loaders. You can specify that an application class loader load the contents of a Web module rather than the Web module class loader.




Each class loader is a child of the previous class loader. That is, the application module class loaders are children of the WebSphere extensions class loader, which is a child of the CLASSPATH Java class loader. Whenever a class needs to be loaded, the class loader usually delegates the request to its parent class loader. If none of the parent class loaders can find the class, the original class loader attempts to load the class. Requests can only go to a parent class loader; they cannot go to a child class loader. If the WebSphere extensions class loader is requested to find a class in a Java EE module, it cannot go to the application module class loader to find that class and a ClassNotFoundException error occurs. After a

class is loaded by a class loader, any new classes that it tries to load reuse the same class loader or go up the precedence list until the class is found.

## Class-loader isolation policies

The number and function of the application module class loaders depend on the class-loader policies that are specified in the server configuration. Class loaders provide multiple options for isolating applications and modules to enable different application packaging schemes to run on an application server.

Two class-loader policies control the isolation of applications and modules:

| Class-loader policy | Description |
| --- | --- |
| **Application** | Application class loaders load EJB modules, dependency JAR files, embedded resource adapters, and application-scoped shared libraries. Depending on the application class-loader policy, an application class loader can be shared by multiple applications (Single) or unique for each application (Multiple). The application class-loader policy controls the isolation of applications that are running in the system. When set to `Single`, applications are not isolated. When set to `Multiple`, applications are isolated from each other. |
| **WAR** | By default, Web module class loaders load the contents of the WEB-INF/classes and WEB-INF/lib directories. The application class loader is the parent of the Web module class loader. You can change the default behavior by changing the Web application archive (WAR) class-loader policy of the application.<br><br>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.<br>**Note:** The console and the underlying deployment.xml file use different names for WAR class-loader policy values. In the console, the WAR class-loader policy values are `Application` or `Module`. However, in the underlying deployment.xml file where the policy is set, the WAR class-loader policy values are `Single` instead of `Application`, or `Multiple` instead of `Module`. `Application` is the same as `Single`, and `Module` is the same as `Multiple`. |

**Note:** WebSphere Application Server class loaders never load application client modules.

For each application server in the system, you can set the application class-loader policy to `Single` or `Multiple`. When the application class-loader policy is set to `Single`, then a single application class loader loads all EJB modules, dependency JAR files, and shared libraries in the system. When the application class-loader policy is set to `Multiple`, then each application receives its own class loader that is used for loading the EJB modules, dependency JAR files, and shared libraries for that application.

An application class loader loads classes from Web modules if the application's WAR class-loader policy is set to `Application`. If the application's WAR class-loader policy is set to `Module`, then each WAR module receives its own class loader.

The following example shows that when the application class-loader policy is set to `Single`, a single application class loader loads all of the EJB modules, dependency JAR files, and shared libraries of all applications on the server. The single application class loader can also load Web modules if an application has its WAR class-loader policy set to `Application`. Applications that have a WAR class-loader policy set to `Module` use a separate class loader for Web modules.

```
Server's application class-loader policy: Single
Application's WAR class-loader policy: Module

Application 1
```

```
 Module:  EJB1.jar
 Module: WAR1.war
   MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = Module
Application 2
 Module:   EJB2.jar
  MANIFEST Class-Path: Dependency2.jar
 Module: WAR2.war
  WAR Classloader Policy = Application
```

**WebSphere extensions class loader**

Classpath:
WebSphere/AppServer/classes
WebSphere/AppServer/lib
WebSphere/AppServer/lib/ext

**Application class loader**

Classpath:
Ejb1.jar
Dependency1.jar
Ejb2.jar
Dependency2.jar
WAR2.war (WEB-INF/classes, ...)

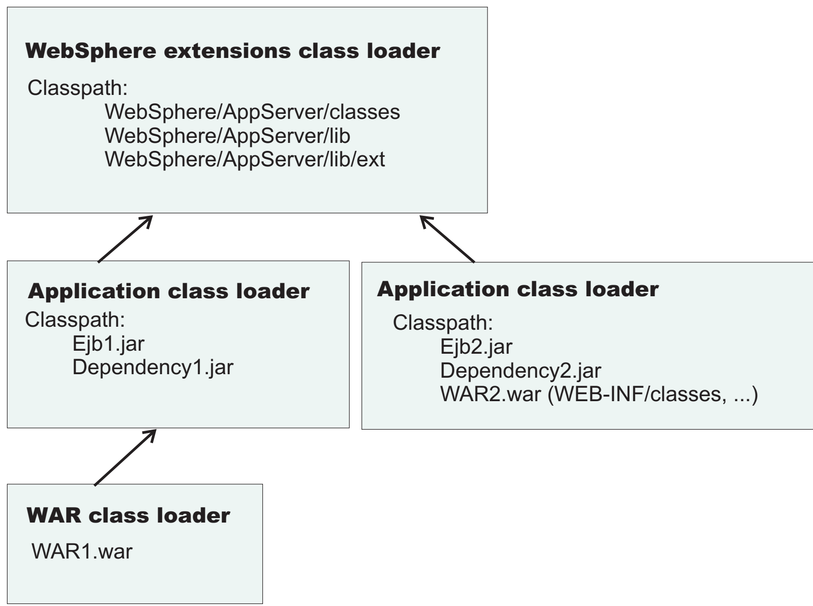
**WAR class loader**

WAR1.war

The following example shows that when the application class-loader policy of an application server is set to `Multiple`, each application on the server has its own class loader. An application class loader also loads its Web modules if the application WAR class-loader policy is set to `Application`. If the policy is set to `Module`, then a Web module uses its own class loader.

```
Server's application class-loader policy: Multiple
Application's WAR class-loader policy: Module

Application 1
 Module:  EJB1.jar
 Module: WAR1.war
  MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = Module
Application 2
 Module:   EJB2.jar
  MANIFEST Class-Path: Dependency2.jar
 Module: WAR2.war
  WAR Classloader Policy = Application
```




## Class-loader modes

The class-loader delegation mode, also known as the *class loader order*, determines whether a class loader delegates the loading of classes to the parent class loader. The following values for class-loader mode are supported:

| Class-loader mode | Description |
|---|---|
| **Parent first**<br><br>Also known as **Classes loaded with parent class loader first**. | The `Parent first` class-loader mode causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. This value is the default for the class-loader policy and for standard JVM class loaders. |
| **Parent last**<br><br>Also known as **Classes loaded with local class loader first** or **Application first**. | The `Parent last` class-loader mode causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader. |

The following settings determine the mode of a class loader:

- If the application class-loader policy of an application server is `Single`, the server-level mode value defines the mode for an application class loader.
- If the application class-loader policy of an application server is `Multiple`, the application-level mode value defines the mode for an application class loader.
- If the WAR class-loader policy of an application is `Module`, the module-level mode value defines the mode for a WAR class loader.

# Configuring class loaders of a server

You can configure the application class loaders for an application server. Class loaders enable applications that are deployed on the application server to access repositories of available classes and resources.

## Before you begin

This topic assumes that an administrator created an application server on a WebSphere Application Server product.

## About this task

Configure the class loaders of an application server to set class-loader policy and mode values which affect all applications that are deployed on the server. Use the administrative console to configure the class loaders.

1. Click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** to access an application server settings page.
2. Specify the application class-loader policy for the application server.

   The application class-loader policy controls the isolation of applications that run in the system (on the server). An application class loader groups enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Dependency JAR files are JAR files that contain code which can be used by both enterprise beans and servlets. The application class-loader policy controls whether an application class loader can be shared by multiple applications or is unique for each application.

   Use the application server settings page to specify the application class-loader policy for the server:

| Option | Description |
|---|---|
| **Single** | Applications are not isolated from each other. Uses a single application class loader to load all of the EJB modules, shared libraries, and dependency JAR files in the system. |
| **Multiple** | Applications are isolated from each other. Gives each application its own class loader to load the EJB modules, shared libraries, and dependency JAR files of that application. |

3. Specify the application class-loader mode for the application server.

   The application class loading mode specifies the class-loader mode when the application class-loader policy is `Single`.

   On the application server settings page, select either of the following values:

| Option | Description |
|---|---|
| **Classes loaded with parent class loader 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. `Classes loaded with parent class loader first` is the default value for class loading mode.<br><br>This value is also known as `parent first`. |
| **Classes loaded with local class loader first (parent last)** | Causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Using this policy, an application class loader can override and provide its own version of a class that exists in the parent class loader. |

4. Specify the class-loader mode for the class loader.
   a. On the application server settings page, click **Java and Process Management** → **Class loader** to access the Class loader page.
   b. On the Class loader page, click **New** to access the settings page for a class loader.

c. On the class loader settings page, specify the class loader order.

   The `Classes loaded with parent class loader first` value causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path.

   The `Classes loaded with local class loader first (parent last)` value causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent.

d. Click **OK**.

An identifier is assigned to a class-loader instance. The instance is added to the collection of class loaders shown on the Class loader page.

### What to do next

Save the changes to the administrative configuration.

## Class loader collection

Use this page to manage class-loader instances on an application server. A class loader determines whether an application class loader or a parent class loader finds and loads Java class files for an application.

To view this administrative console page, click **Servers** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Java and Process Management** → **Class loader**.

## Class loader ID

Specifies a string that is unique to the server identifying the class-loader instance. The product assigns the identifier.

## Class loader order

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for development kit class loaders and WebSphere Application Server class loaders is `Classes loaded with parent class loader first (Parent first)`. By specifying `Classes loaded with local class loader first (Parent last)`, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

## 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** → **Server Types** → **WebSphere application servers** → ***server_name*** → **Java and Process Management** → **Class loader** → **class_loader_ID**.

### Class loader ID

Specifies a string that is unique to the server identifying the class-loader instance. The product assigns the identifier.

| | |
|---|---|
| **Data type** | String |

### Class loader order

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for development kit class loaders and WebSphere Application Server class loaders is `Classes loaded with parent class loader first`. By specifying `Classes loaded with local class loader first (parent last)`, your application can override classes contained in the parent

class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

The options are `Classes loaded with parent class loader first` and `Classes loaded with local class loader first (parent last)`. The default is to search in the parent class loader before searching in the application class loader to load a class.

For your application to use the default configuration of Jakarta Commons Logging in this product, set this application class loader order to `Classes loaded with parent class loader first`. For your application to override the default configuration of Jakarta Commons Logging, your application must provide the configuration in a form supported by Jakarta Commons Logging and this class loader order must be set to `Classes loaded with local class loader first (parent last)`. Also, to override the default configuration, set the class loader order for each Web module in your application so that the correct logger factory loads.

| | |
|---|---|
| **Data type** | String |
| **Default** | Parent first |

## Configuring application class loaders

You can set values that control the class-loading behavior of an installed enterprise application. Class loaders enable an application to access repositories of available classes and resources.

### Before you begin

This topic assumes that you installed an application on an application server.

### About this task

Configure the class loaders of an enterprise application to set class-loader policy and mode values for this application.

An application class loader groups enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Dependency JAR files are JAR files that contain code which can be used by both enterprise beans and servlets.

An application class loader is the parent of a Web application archive (WAR) class loader. By default, a Web module has its own WAR class loader to load the contents of the Web module. The WAR class-loader policy value of an application class loader determines whether the WAR class loader or the application class loader is used to load the contents of the Web module.

Use the administrative console to configure the class loaders.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Class loading and update detection** to access the settings page for an application class loader.
2. Specify whether to reload application classes when the application or its files are updated.

   By default, class reloading is not enabled. Select **Override class reloading settings for Web and EJB modules** to choose to reload application classes. You might specify different values for EJB modules and for Web modules such as servlets and JavaServer Pages (JSP) files.

3. Specify the number of seconds to scan the application's file system for updated files.

   The value specified for **Polling interval for updated files** takes effect only if class reloading is enabled. The default is the value of the reloading interval attribute in the IBM extension (META-INF/ibm-application-ext.xmi) file of the enterprise application (EAR file). You might specify different values for EJB modules and for Web modules such as servlets and JSP files.

   To enable reloading, specify an integer value that is greater than zero (for example, 1 to 2147483647).

   To disable reloading, specify zero (0).

4. Specify the class loader order for the application.

   The application class loader order specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The default is to search in the parent class loader before searching in the application class loader to load a class.

   Select either of the following values for **Classes loader order**:

| Option | Description |
| --- | --- |
| **Classes loaded with parent class loader first** | Causes the class loader to search in the parent class loader first to load a class. This value is the standard for Development Kit class loaders and WebSphere Application Server class loaders. |
| **Classes loaded with local class loader first (parent last)** | Causes the class loader to search in the application class loader first to load a class. By specifying `Classes loaded with local class loader first (parent last)`, your application can override classes contained in the parent class loader.<br>**Note:** Specifying the `Classes loaded with local class loader first (parent last)` value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes. |

5. Specify whether to use a single or multiple class loaders to load Web application archives (WAR files) of your application.

   By default, Web modules have their own WAR class loader to load the contents of the WEB-INF/classes and WEB-INF/lib directories. The default WAR class loader value is `Class loader for each WAR file in application`, which uses a separate class loader to load each WAR file. Setting the value to `Single class loader for application` causes the application class loader to load the Web module contents as well as the EJB modules, shared libraries, RAR files, and dependency JAR files associated to the application. The application class loader is the parent of the WAR class loader.

   Select either of the following values for **WAR class loader policy**:

| Option | Description |
| --- | --- |
| **Class loader for each WAR file in application** | Uses a different class loader for each WAR file. |
| **Single class loader for application** | Uses a single class loader to load all of the WAR files in your application. |

6. Click **OK**.

### What to do next

Save the changes to the administrative configuration.

## Configuring Web module class loaders

You can set values that control the class-loading behavior of an installed Web module.

## Before you begin

This topic assumes that you installed a Web module on an application server.

## About this task

Configure the class loader order value of an installed Web module. By default, a Web module has its own Web application archive (WAR) class loader to load the contents of the Web module, which are in the WEB-INF/classes and WEB-INF/lib directories.

An application class loader is the parent of a WAR class loader. The WAR class-loader policy value of an application class loader determines whether the WAR class loader or the application class loader is used to load the contents of the Web module.

The default WAR class loader policy value is `Class loader for each WAR file in application`. If the policy is set to `Class loader for each WAR file in application`, then each Web module receives its own class loader whose parent is the application class loader. If the policy is set to `Single class loader for application` on the settings page of an application class loader, then the application class loader loads the Web module contents as well as the enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Thus, the configuration of the parent application class loader affects the WAR class loader.

Use the administrative console to configure the application and WAR class loaders.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. If you have not done so already, configure the application class loader.

   Settings such as **Override class reloading settings for Web and EJB modules**, **Polling interval for updated files** and **WAR class loader policy** can affect Web module class loading.

   If **WAR class loader policy** is set to `Class loader for each WAR file in application`, then the Web module receives its own class loader and the WAR class-loader policy of the Web module defines the mode for a WAR class loader. If the policy is set to `Single class loader for application`, then the application class loader loads the Web module contents.

2. Specify the class loader order for the installed Web module.

   The Web module class-loader mode specifies whether the class loader searches in the parent application class loader or in the WAR class loader first to load a class. The default is to search in the parent application class loader before searching in the WAR class loader to load a class.

Select either of the following values for **Class loader order**:

| Option | Description |
| --- | --- |
| **Classes loaded with parent class loader first** | Causes the class loader to search in the parent application class loader first to load a class. This is the standard for Development Kit class loaders and WebSphere Application Server class loaders. **Note:** If classes and resources needed by the Web module cannot be accessed by the application class loader, but can be accessed by the WAR class loader, specify `Classes loaded with local class loader first (parent last)`. If the application class loader cannot find a class, the class loader delegates the request to find the class to its parent, the WebSphere Application Server extensions class loader. If the WebSphere Application Server extensions class loader cannot find the class, the class loader delegates the request to its parent, the bootstrap, extensions, and CLASSPATH class loaders created by the Java virtual machine. Requests can only go to a parent class loader; they cannot go to a child class loader. Thus, if `Classes loaded with parent class loader first` is specified, the WAR class loader never receives a request to load a class. |
| **Classes loaded with local class loader first (parent last)** | Causes the class loader to search in the WAR class loader first to load a class. By specifying `Classes loaded with local class loader first (parent last)`, your WAR class loader can override classes contained in the parent application class loader. **Note:** Specifying the `Classes loaded with local class loader first (parent last)` value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes. |

3. Click **OK**.

### What to do next

Save the changes to the administrative configuration.

## Class loading: Resources for learning

Additional information and guidance on class loading is available on various Internet sites.

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

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

View links to additional information about:

- “Programming model and decisions” on page 28
- “Programming instructions and examples” on page 28
- “Programming specifications” on page 28

## Programming model and decisions

- Demystifying class loading problems, Part 1: An introduction to class loading and debugging tools - Learn how class loading works and how your JVM can help you sort out class loading problems (*developerWorks*, November 2005), http://www.ibm.com/developerworks/java/library/j-dclp1/?S_TACT=106AH10W&S_CMP=NC
- Demystifying class loading problems, Part 2: Basic class loading exceptions - An in-depth look at some simple class loading quirks and conundrums (*developerWorks*, December 2005), http://www.ibm.com/developerworks/java/library/j-dclp2.html?S_TACT=105AGX10&S_CMP=NC
- Demystifying class loading problems, Part 3: Tackling more unusual class loading problems - Understand class loading and quash subtle exceptions (*developerWorks*, December 2005), http://www.ibm.com/developerworks/java/library/j-dclp3/?S_TACT=105AGX10&S_CMP=NC
- J2EE Class Loading Demystified (*developerWorks*, August 2002), http://www.ibm.com/developerworks/websphere/library/techarticles/0112_deboer/deboer.html
- Java programming dynamics, Part 1: Classes and class loading - A look at classes and what goes on as they're loaded by a JVM (*developerWorks*, April 2003), http://www.ibm.com/developerworks/java/library/j-dyn0429/

## Programming instructions and examples

- *WebSphere Application Server V6.1:* System Management Configuration Handbook, SG24-7304-00, http://www.redbooks.ibm.com/abstracts/SG247304.html?Open
- *IBM WebSphere Developer Technical Journal:* Co-hosting multiple versions of J2EE applications, http://www.ibm.com/developerworks/websphere/techjournal/0405_poddar/0405_poddar.html

## Programming specifications

- Specifications and API documentation

# Chapter 5. Deploying and administering enterprise applications

Deploying an enterprise application file consists of installing an application file on a server configured to hold installable Java Platform, Enterprise Edition (Java EE) modules.

## Before you begin

Before installing an enterprise application or other installable module on an application server, you must develop the module, assemble the module, and configure the target server . Before choosing a deployment target for the module, ensure that the target version is compatible with your module.

## About this task

During installation, you can configure the module enough to enable it to run on the server. After installation, you can configure the module further, start or stop the application, and otherwise manage its activity.

The topics in this section describe how to deploy and administer applications or modules using the administrative console. You can also use scripting or administrative programs (JMX).

- Install Java EE application files on an application server.
- Edit the administrative configuration for an application.
- Optional: View the deployment descriptor for an application or module.
- Start and stop enterprise applications.
- Export enterprise applications.
- Export a file in a Java EE application or module.
- Export DDL files.
- Update a Java EE application or module.
- Uninstall enterprise applications.
- Remove a file from a Java EE application or module.

## What to do next

After making changes to administrative configurations of your applications in the administrative console, ensure that you save the changes.

# Enterprise (Java EE) applications

Enterprise applications (or Java EE applications) are applications that conform to the Java Platform, Enterprise Edition (Java EE) specification. Prior to Java EE 5, the specification name was Java 2 Platform, Enterprise Edition (J2EE). The term *Java EE* includes Java EE 5 and J2EE specifications.

Enterprise applications can consist of the following:

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

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

© Copyright IBM Corp. 2008

## System applications

A *system application* is a Java Platform, Enterprise Edition (Java EE) enterprise application that is central to a WebSphere Application Server product.

Examples of system applications include *isclite*, *managementEJB* and *filetransfer*.

Because a system application is an important part of a WebSphere Application Server product, a system application is deployed when the product is installed and is updated only through a product fix or upgrade. For some system applications, such as *filetransfer*, users cannot change the metadata for the system application, unless the metadata assigns users and groups for security purposes. For these applications, non-security related metadata such as its Java EE bindings or extensions must be updated through a product fix or upgrade.

System applications are not shown in the list of installed applications on the console Enterprise Applications page, or through wsadmin and Java application programming interfaces, to prevent users from accidentally stopping, updating or removing the system applications.

Note that Java EE Samples are not system applications even though they are provided as part of a WebSphere Application Server product. Similarly, applications that support changes to their metadata are not system applications.

## Common deployment framework

The *common deployment framework* enables you to implement plug-ins that add steps to default Java Platform, Enterprise Edition (Java EE) application management operations such as install, uninstall, edit and update.

Using the framework, you can implement management operations on specific types of deployable contents. For example, the deployable contents might include EAR, WAR, JAR or other Java EE modules and the management operations might include install and uninstall. Each operation is divided into a number of steps. For example, the install operation has steps for EJBDeploy and JavaServer Pages (JSP) compilation, among others. Using the common deployment framework, you can add steps to the default logic for Java EE operations.

The product supports framework plug-ins that extend deployment of EAR files. An EAR file has operations such as createEarWrapper, installApplication, uninstallApplication and editApplication. Using a framework plug-in, you can add steps to default install operations that support, for example, creating additional configuration artifacts in a configuration session, modifying an input EAR file using code generation, or additional validating of input parameters.

To extend application management operations using the framework, a plug-in must do the following:

- Implement each step.

  A *step* runs logic that performs an operation. A step can access the deployment context and the deployable object. The *deployment context* provides information such as the operation name, the configuration session identifier, the temporary location for creating temporary files, operations parameters, and the like. A step is added by the extension provider.
- Implement an extension provider that adds each implemented step.

  An *extension provider* is a class that provides steps for an operation on a given type, the EAR file type.
- Register the plug-in with a WebSphere Application Server server.

  The plug-in is implemented as an Eclipse plug-in and is placed in *app_server_root*/`plugins` directory. Add the extension point for the extension provider in the `META-INF/plugin.xml` file within the plug-in JAR file.

For an example of these steps, refer to "Extending application management operations through programming" on page 153.

## Installing enterprise application files

As part of deploying an application, you install application files on a server configured to hold installable modules.

### Before you begin

Before you can install your Java Platform, Enterprise Edition (Java EE) application files on an application server, you must assemble modules as needed.

Also, before you install the files, configure the target application server. As part of configuring the server, determine whether your application files can be installed to your deployment targets.

### About this task

You can install the following modules on a server:
- Enterprise archive (EAR)
- Enterprise bean (EJB)
- Web archive (WAR)
- Session Initiation Protocol (SIP) module (SAR)
- Resource adapter (connector or RAR)
- Application client modules

Application client files can be installed in a WebSphere Application Server configuration but cannot be run on a server.

Complete the following steps to install your files.
1. Determine which method to use to install your application files. The product provides several ways to install modules.
2. Install the application files using
   - Administrative console
   - wsadmin scripts
   - Java administrative programs that use Java Management Extensions (JMX) application programming interfaces (APIs)
   - Java programs that define a Java EE DeploymentManager object in accordance with Java EE Application Deployment specification (JSR-88)
3. Start the deployed application files using
   - Administrative console
   - wsadmin startApplication
   - Java programs that use ApplicationManager or AppManagement MBeans
   - Java programs that define a Java EE DeploymentManager object in accordance with Java EE Application Deployment specification (JSR-88)

### What to do next

Save the changes to your administrative configuration.

Next, test the application. For example, point a Web browser at the URL for a deployed application (typically http://*hostname*:9060/*Web_module_name*, where *hostname* is your valid Web server and 9060 is the default port number) and examine the performance of the application. If the application does not perform as desired, edit the application configuration, then save and test it again.

# Installable enterprise module versions

The contents of a Java Platform, Enterprise Edition (Java EE) module affect whether you can install the module on a deployment target. A *deployment target* is a server on a WebSphere Application Server product.

## Installable application modules

Select only appropriate deployment targets for a module. You must install an application, enterprise bean (EJB) module, Session Initiation Protocol (SIP) module (SAR), or Web module on a Version 7.x target under any of the following conditions:

- The module supports Java Platform, Enterprise Edition (Java EE) 5.
- The module calls a 7.x runtime application programming interface (API).
- The module uses a 7.x product feature.

If a module supports Java 2 Platform, Enterprise Edition (J2EE) 1.4, then you must install the module on a Version 6.x or 7.x deployment target. Modules that call a 6.1.x API or use a 6.1.x feature can be installed on a 6.1.x or 7.x deployment target. Modules that call a 6.0.x API or use a 6.0.x feature can be installed on a 6.0.x, 6.1.x or 7.x deployment target. Modules that require 6.1.x feature pack functionality can be installed on a 7.x deployment target or on a 6.1.x deployment target that has been enabled with that feature pack.

Selecting options such as **Precompile JavaServer Pages files**, **Use binary configuration**, **Deploy Web services** or **Deploy enterprise beans** during application installation indicates that the application uses 6.1.x product features. You cannot deploy such applications on a 5.x or 6.0.x deployment target. You must deploy such applications on a 6.1.x or 7.x deployment target.

You can install an application or module developed for a Version 5.x product on a 5.x, 6.x or 7.x deployment target.

**Note:** You must package container-managed persistence (CMP) or bean-managed persistence (BMP) entity beans in an EJB 2.1 or earlier module. You cannot install an EJB 3.0 module that contains CMP or BMP entity beans. Installation fails when a CMP or BMP entity bean is packaged in an EJB 3.0 module. You can install EJB 2.1 or earlier modules on a 5.x, 6.x or 7.x deployment target.

## Installable RAR files

You can install a standalone resource adapter (connector) module, or RAR file, developed for a Version 5.x product to a 5.x, 6.x or 7.x deployment target, provided the module does not call any 6.x or 7.x runtime APIs. If the module calls a 6.x API, then you must install the module on a 6.x or 7.x deployment target. If the module calls a 7.x API, then you must install the module on a 7.x deployment target.

## Deployment targets

A *5.x deployment target* is a server on a WebSphere Application Server Version 5 product.

A *6.x deployment target* is a server on a WebSphere Application Server Version 6 product.

A *7.x deployment target* is a server on a WebSphere Application Server Version 7 product.

*Table 2. Compatible deployment target versions for 5.x, 6.x and 7.x modules*

| Module type | Module Java support | Module calls 6.x or 7.x runtime APIs or uses 6.x or 7.x features? | Client versions that can install module | Deployment target versions |
|---|---|---|---|---|

*Table 2. Compatible deployment target versions for 5.x, 6.x and 7.x modules (continued)*

| | | | | |
|---|---|---|---|---|
| Application, EJB, Web, or client | J2EE 1.3 | No | 5.x, 6.x or 7.x | 5.x, 6.x or 7.x |
| Application, EJB, Web, or client | J2EE 1.3 | Yes | 6.x for 6.x or 7.x APIs or features<br><br>7.x for 7.x APIs or features | 6.x or 7.x<br><br>You must install modules that call 6.1.x runtime APIs or use 6.1.x features on a 6.1.x or 7.x deployment target. You can install modules that call 6.0.x runtime APIs or use 6.0.x features on any 6.x or 7.x deployment target. |
| Application, EJB, SAR, Web, or client | J2EE 1.4 | Yes or No | 6.x or 7.x | 6.x or 7.x |
| Application, EJB, SAR, Web, or client | Java EE 5 | Yes or No | 7.x | 7.x |
| Resource adapter | JCA 1.0 | No | 5.x, 6.x or 7.x | 5.x, 6.x or 7.x |
| Resource adapter | JCA 1.0 | Yes | 6.x or 7.x | 6.x or 7.x<br><br>You must install modules that call 6.1.x runtime APIs on a 6.1.x or 7.x deployment target. You can install modules that call 6.0.x runtime APIs on any 6.x or 7.x deployment target. |
| Resource adapter | JCA 1.5 | Yes or No | 6.x or 7.x | 6.x or 7.x<br><br>You must install modules that call 6.1.x runtime APIs on a 6.1.x or 7.x deployment target. You can install modules that call 6.0.x runtime APIs on any 6.x or 7.x deployment target. |

## Ways to install enterprise applications or modules

The product provides several ways to install Java Platform, Enterprise Edition (Java EE) application files.

Installable files include enterprise archive (EAR), enterprise bean (EJB), Web archive (WAR), Session Initiation Protocol (SIP) module (SAR), resource adapter (connector or RAR), and application client modules. They can be installed on a server. Application client files can be installed in a WebSphere Application Server configuration but cannot be run on a server.

*Table 3. Ways to install application files*

| Option | Method | Modules | Comments | Starting after install |
|---|---|---|---|---|
| Administrative console install wizard<br><br>See "Installing enterprise application files with the console" on page 36. | Click **Applications** → **New application** → **New Enterprise Application** in the console navigation tree and follow instructions in the wizard. | Files for all of the following modules:<br>• EAR<br>• EJB<br>• WAR<br>• SAR<br>• RAR<br>• Application client | Provides one of the easier ways to install application files. See "Preparing for application installation settings" on page 42 for guidance.<br><br>For applications that do not require changes to the default bindings, after you specify the application file, expand **Choose to generate default bindings and mappings**, select **Generate default bindings**, click the Summary step, and then click **Finish**. | Click **Start** on the Enterprise applications page accessed by clicking **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree. |
| wsadmin scripts | Invoke AdminApp object install commands in a script or at a command prompt. | Files for all of the following modules:<br>• EAR<br>• EJB<br>• WAR<br>• SAR<br>• RAR<br>• Application client | ″Getting started with scripting″ in the Using the administrative clients PDF provides an overview of wsadmin. | • Invoke the AdminApp startApplication command.<br><br>• Invoke the startApplication method on an ApplicationManager MBean using AdminControl. |
| Java application programming interfaces | Install programs by completing the steps in ″Installing an application through programming″ in the Using the administrative clients PDF. | All EAR files | Use MBeans to install the application. ″Managing applications through programming″ in the Using the administrative clients PDF provides an overview of Java MBean programming. | Start the application by calling the startApplication method on a proxy. |

*Table 3. Ways to install application files (continued)*

| Option | Method | Modules | Comments | Starting after install |
|---|---|---|---|---|
| Rapid deployment tools<br><br>Refer to topics under **Rapid deployment of J2EE applications**. | Briefly, do the following:<br><br>1. Update your J2EE application files.<br><br>2. Set up the rapid deployment environment.<br><br>3. Create a free-form project.<br><br>4. Launch a rapid deployment session.<br><br>5. Drop your updated application files into the free-form project. | J2EE modules at the J2EE 1.3 or 1.4 specification levels, including EAR files and the following stand-alone modules:<br>• EJB<br>• WAR<br>• SAR<br>• RAR<br>• Application client<br><br>The rapid deployment tools do not support the Java EE 5.0 or J2EE 1.2 specification levels. | Rapid deployment tools offer the following advantages:<br><br>• You do not need to assemble your J2EE application files prior to deployment.<br><br>• You do not need to use other installation tools mentioned in this table to deploy the files. | Use any of the above options to start the application. Clicking **Start** on the Enterprise applications page is the easiest option. |
| Java programs | Code programs that use Java EE DeploymentManager (JSR-88) methods. | All Java EE modules, including EAR files and the following stand-alone modules:<br>• EJB<br>• WAR<br>• SAR<br>• RAR<br>• Application client | • Uses Java EE Application Deployment Specification (JSR-88).<br><br>• Can customize modules using DConfigBeans. | Call the Java EE DeploymentManager (JSR-88) start method in a program to start the deployed modules when the module's running environment initializes. |

**Note:** In the Version 6.1 Feature Pack for Web services and Feature Pack for EJB 3.0, the default is to scan pre-Java EE 5 Web application modules to identify JAX-WS services and to scan pre-Java EE 5 Web application modules and EJB modules for service clients during application installation. For Version 7.0, the default is not to scan pre-Java EE 5 modules for annotations during application installation or server startup. To preserve backward compatibility with either or both feature packs, you can define Java virtual machine custom properties on servers to request scanning during application installation and server startup.

- You can define these custom properties using the console. Click **Servers** → **Server Types** → **WebSphere application servers** → ***server name*** → **Java and Process Management** → **Process definition** → **Java virtual machine** → **Custom properties**. To request scanning for Feature Pack for Web services modules, set the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` custom property to `true`. To request scanning for Feature Pack for EJB 3.0 modules, set the `com.ibm.websphere.ejb.UseEJB61FEPScanPolicy` custom property to `true`. The default value for each of these custom properties is `false`. You must change the setting on each server that requires a change in the default.
- You can specify values for these custom properties in the META-INF/MANIFEST.MF file of a module. Values specified in the META-INF/MANIFEST.MF file always take precedence over a server-level setting.
- When using the launchClient tool to run the application client, you need to define these custom properties using the `-CCD` option. For example, `launchClient app.ear -CCD-CCDcom.ibm.websphere.ejb.UseEJB61FEPScanPolicy=true`.

## Installing enterprise application files with the console

Installing Java Platform, Enterprise Edition (Java EE) application files consists of placing assembled enterprise application, Web, enterprise bean (EJB), or other installable modules on a server or cluster configured to hold the files. Installed files that start and run properly are considered *deployed*.

### Before you begin

Before installing enterprise application files, ensure that you are installing your application files onto a compatible deployment target. If the deployment target is not compatible, select a different target.

Optionally, determine whether the application that you are installing uses library files that other deployed applications also use. You can define a shared library for each of these shared files. Using shared libraries reduces the number of library file copies on your workstation or server.

### About this task

To install new enterprise application files to a WebSphere Application Server configuration, you can use the following options:
- Administrative console
- wsadmin scripts
- Java MBean programs
- Java programs that call Java EE DeploymentManager (JSR-88) methods

This topic describes how to use the administrative console to install an application, EJB component, Session Initiation Protocol (SIP) module (SAR), or Web module.

**Note:** After you start performing the steps below, click **Cancel** to exit if you decide not to install the application. Do not simply move to another administrative console page without first clicking **Cancel** on an application installation page.

1. Click **Applications** → **New application** → **New Enterprise Application** in the console navigation tree.
2. On the first Preparing for application installation page:
   a. Specify the full path name of the source enterprise application file (.ear file otherwise known as an *EAR file*).

      The EAR file that you are installing can be either on the client machine (the machine that runs the Web browser) or on the server machine (the machine to which the client is connected). If you specify an EAR file on the client machine, then the administrative console uploads the EAR file to the machine on which the console is running and proceeds with application installation.

      You can also specify a stand-alone Web application archive (WAR), SAR, or Java archive (JAR) file for installation.

      If the EAR file resides on the server machine, and the server is an iSeries® server, ensure that user profile QEJBSVR has *R authority to the EAR file and at least *X authority to all the directories in the path containing the EAR file.

   b. Click **Next**.
3. On the second Preparing for application installation page:
   a. Select whether to view all installation options.

      **Fast Path - Prompt only when additional information is required**
      : Displays the module mapping step as well as any steps that require you to specify needed information to install the application successfully.

      **Detailed - Show all installation options and parameters**
      : Displays all installation options.

   b. Select whether to generate default bindings.

Select **Generate default bindings** to have the product supply default values for incomplete Java Naming and Directory (JNDI) and other bindings in the application. The product does not change existing bindings.

You do not need to specify JNDI values for EJB bean, local home, remote home, or business interfaces of EJB 3.0 modules. The product assigns container default values during run time. Similarly, for any EJB reference within an EJB 3.0, Web 2.4, or Web 2.5 module, you do not need to specify JNDI values because the product resolves the targets automatically during run time. Even when you select **Generate default bindings**, the product does not generate default values for those JNDI values but it does generate default values for other bindings such as virtual host.

You can customize default values used in generating default bindings. "Preparing for application installation binding settings" on page 42 describes available customizations and provides sample bindings.

c. Click **Next**. If security warnings are displayed, click **Continue**. The Install New Application pages are displayed. If you chose to generate default bindings, you can proceed to the Summary step. "Example: Installing an EAR file using the default bindings" on page 60 provides sample steps.

4. Specify values for installation options as needed.

   You can click on a step number to move directly to that panel instead of clicking **Next**. The contents of the application or module that you are installing determines which panels are available.

| Panel | Description |
| --- | --- |
| **Select installation options** | On the **Select installation options** panel, provide values for the settings specific to the product. Default values are used if you do not specify a value. |
| **Map modules to servers** | On the **Map modules to servers** panel, specify deployment targets where you want to install the modules contained in your application. Modules can be installed on the same deployment target or dispersed among several deployment targets. Each module must be mapped to a target server.<br><br>On single-server products, a deployment target can be an application server or Web server. |
| **Provide options to compile JSPs** | If the **Precompile JavaServer Pages files** setting is enabled on the **Select installation options** panel and your application uses JavaServer Pages (JSP) files, then you can specify JSP compiler options on the **Provide options to compile JSPs** panel. |
| **Provide JNDI names for beans** | On the **Provide JNDI names for beans** panel, specify a JNDI name for each enterprise bean in every EJB 2.1 and earlier module. You must specify a JNDI name for every enterprise bean defined in the application. For example, for the EJB module MyBean.jar, specify `MyBean`.<br><br>As to EJB 3.0 modules, you can specify JNDI names, local home JNDI names, remote home JNDI names, or no JNDI names. If you do not specify a value, the product provides a default value. |
| **Bind EJB business** | On the **Bind EJB business** panel, you can specify business interface JNDI names for EJB 3.0 modules. If you specified a JNDI name for a bean on the **Provide JNDI names for beans** panel, do not specify a business interface JNDI name on this panel for the same bean. If you do not specify the JNDI name for a bean, you can optionally specify a business interface JNDI name. When you do not specify a business interface JNDI name, the product provides a container default. |
| **Map default data sources for modules containing 1.x entity beans** | If your application uses EJB modules that contain Container Managed Persistence (CMP) beans that are based on the EJB 1.x specification, for **Map default data sources for modules containing 1.x entity beans**, specify a JNDI name for the default data source for the EJB modules. The default data source for the EJB modules is optional if data sources are specified for individual CMP beans. |

| Panel | Description |
| --- | --- |
| **Map EJB references to beans** | On the **Map EJB references to beans** panel, if your application defines EJB references, you can specify JNDI names for enterprise beans that represent the logical names specified in EJB references.<br><br>If the EJB reference is from EJB 3.0, Web 2.4, or Web 2.5 module, the JNDI name is optional. For earlier modules, each EJB reference defined in the application must be bound to an EJB file.<br><br>If **Allow EJB reference targets to resolve automatically** is enabled, the JNDI name is optional for all modules. The product provides a container default value or automatically resolves the EJB reference for incomplete bindings. |
| **Map resource references to resources** | If your application defines resource references, for **Map resource references to resources**, specify JNDI names for the resources that represent the logical names defined in resource references. You can optionally specify login configuration name and authentication properties for the resource. After specifying authentication properties, click **OK** to save the values and return to the mapping step. You can optionally specify extended data source properties to enable a data source that uses heterogeneous pooling to connect to a DB2® database. 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. |
| **Map virtual hosts for Web modules** | If your application uses Web modules, for **Map virtual hosts for Web modules**, select a virtual host from the list that should map to a Web module defined in the application. The port number specified in the virtual host definition is used in the URL that is used to access artifacts such as servlets and JSP files in the Web module. Each Web module must have a virtual host to which it maps. Not specifying all needed virtual hosts will result in a validation error displaying after you click **Finish** on the Summary panel. |
| **Map security roles to users or groups** | If the application has security roles defined in its deployment descriptor then, for **Map security roles to users or groups**, specify users and groups that are mapped to each of the security roles. Select **Role** to select all of the roles or select individual roles. For each role, you can specify whether predefined users such as **Everyone** or **All authenticated users** are mapped to it. To select specific users or groups from the user registry:<br>1. Select a role and click **Lookup users** or **Lookup groups**.<br>2. On the Lookup users or groups panel displayed, enter search criteria to extract a list of users or groups from the user registry.<br>3. Select individual users or groups from the results displayed.<br>4. Click **OK** to map the selected users or groups to the role selected on the **Map security roles to users or groups** panel. |
| **Map RunAs roles to users** | If the application has Run As roles defined in its deployment descriptor, for **Map RunAs roles to users**, specify the Run As user name and password for every Run As role. Run As roles are used by enterprise beans that must run as a particular role while interacting with another enterprise bean. Select **Role** to select all of the roles or select individual roles. After selecting a role, enter values for the user name, password, and verify password and click **Apply**. |
| **Ensure all unprotected 1.x methods have the correct level of protection** | If your application contains EJB 1.x CMP beans that do not have method permissions defined for some of the EJB methods, for **Ensure all unprotected 1.x methods have the correct level of protection**, specify if you want to leave such methods unprotected or assign protection with deny all access. |
| **Bind listeners for message-driven beans** | If your application contains message driven enterprise beans, for **Bind listeners for message-driven beans**, provide a listener port name or an activation specification JNDI name for every message driven bean. |

| Panel | Description |
| --- | --- |
| **Map default data sources for modules containing 2.x entity beans** | If your application uses EJB modules that contain CMP beans that are based on the EJB 2.x specification, for **Map default data sources for modules containing 2.x entity beans**, specify a JNDI name for the default data source and the type of resource authorization to be used for the default data source for the EJB modules. You can optionally specify a login configuration name and authentication properties for the data source. When creating authentication properties, you must click **OK** to save the values and return to the mapping step. You can optionally specify extended data source properties to enable a data source that uses heterogeneous pooling to connect to a DB2 database. The default data source for EJB modules is optional if data sources are specified for individual CMP beans. |
| **Map data sources for all 2.x CMP beans** | If your application has CMP beans that are based on the EJB 2.x specification, on the **Map data sources for all 2.x CMP beans** panel, for each of the 2.x CMP beans specify a JNDI name and the type of resource authorization for data sources to be used.<br><br>You can optionally specify a login configuration name and authentication properties for the data source. When creating authentication properties, you must click **OK** to save the values and return to the mapping step. The data source attribute is optional for individual CMP beans if a default data source is specified for the EJB module that contains CMP beans. If neither a default data source for the EJB module nor a data source for individual CMP beans are specified, then a validation error is displayed after you click **Finish** and installation is cancelled. |
| **Ensure all unprotected 2.x methods have the correct level of protection** | If your application contains EJB 2.x CMP beans that do not have method permissions defined in the deployment descriptors for some of the EJB methods, on the **Ensure all unprotected 2.x methods have the correct level of protection** panel, specify whether you want to assign a specific role to the unprotected methods, add the methods to the exclude list, or mark them as unchecked. Methods added to the exclude list are marked as uncallable. For methods marked unchecked no authorization check is performed prior to their invocation. |
| **Provide options to perform the EJB Deploy** | If the **Deploy enterprise beans** setting is enabled on the **Select installation options** panel, then you can specify options for the EJB deployment tool on the **Provide options to perform the EJB Deploy** panel. On this panel, you can specify extra class paths, RMIC options, database types, and database schema names to be used while running the EJB deployment tool.<br><br>You can specify the EJB deployment tool options on this panel when installing or updating an application that contains EJB modules. The EJB deployment tool runs during installation of EJB 1.x or 2.x modules. The EJB deployment tool does not run during installation of EJB 3.0 modules. |
| **Map shared libraries** | On the **Shared library references** and **Shared library mapping** panels, specify shared library files for your application or Web modules to use. A defined shared library must exist to associate your application or module to the library file. |
| **Map shared library relationships** | On the **Map shared library relationships** panel, specify relationship identifiers and composition unit names for shared libraries that modules in your enterprise application reference.<br><br>When installing your enterprise application, the product creates a composition unit for each shared library relationship in the business-level application that you specified for **Business-level application name** on the **Select installation options** panel. |
| **Provide JSP reloading options for Web modules** | If your application uses Web modules, for **Provide JSP reloading options for Web modules**, configure the class reloading of JavaServer Pages (JSP) files. |

| Panel | Description |
| --- | --- |
| **Map context roots for Web modules** | If your application uses Web modules that are defined in the application XML deployment descriptor, for **Map context roots for Web modules**, specify a context root for each Web module in the application.<br><br>The product does not include Web modules from annotations on this panel. |
| **Initialize parameters for servlets** | If your application uses Web modules that support Servlet 2.5, for **Initialize parameters for servlets**, specify or override initial parameters that are passed to the init method of Web module servlet filters.<br><br>This panel shows servlets from the module XML deployment descriptor. Servlet deployment information from annotations is not available on this panel. |
| **Map environment entries for Web modules** | If your application uses Web modules that support Servlet 2.5, for **Map environment entries for Web modules**, configure the environment entries of Web modules such as servlets and JSP files.<br><br>This panel shows environment entries from the module XML deployment descriptor. Environment entry deployment information from annotations is not available in this panel. |
| **Map resource environment entry references to resources** | If your application contains resource environment references, for **Map resource environment entry references to resources**, specify JNDI names of resources that map to the logical names defined in resource environment references. If each resource environment reference does not have a resource associated with it, after you click **Finish** a validation error is displayed. |
| **Correct use of system identity** | If your application defines **Run-As Identity** as *System Identity*, for **Correct use of system identity**, you can optionally change it to *Run-As role* and specify a user name and password for the Run As role specified. Selecting *System Identity* implies that the invocation is done using the WebSphere Application Server security server ID and should be used with caution as this ID has more privileges. |
| **Correct isolation levels for all resource references** | If your application has resource references that map to resources that have an Oracle database doing backend processing, for **Correct isolation levels for all resource references**, specify or correct the isolation level to be used for such resources when used by the application. Oracle databases support ReadCommitted and Serializable isolation levels only. |
| **Bind message destination references to administered objects** | If your application uses message driven beans, for **Bind message destination references to administered objects**, specify the JNDI name of the J2C administered object to bind the message destination reference to the message driven beans.<br><br>If the message destination reference is from a EJB 3.0 module, then the JNDI name is optional and the run time provides a container default value.<br>**Note:** If multiple message destination references link to the same message destination, only one JNDI name is collected. When a message destination reference links to the same message destination as a message driven bean and the destination JNDI name has been collected already, the destination JNDI name for the message destination reference is not collected. |
| **Provide JNDI names for JCA objects** | If your application contains an embedded .rar file, for **Provide JNDI names for JCA objects**, specify the name and JNDI name of each J2C connection factory, J2C administered object and J2C activation specification. |
| **Bind J2C activationspecs to destination JNDI names** | If your application contains an embedded .rar file, its activationSpec property has the value `Destination`, and its introspected type is `javax.jms.Destination`, for **Bind J2C activationspecs to destination JNDI names**, specify the jndiName value for each activation bound to it. |

| Panel | Description |
| --- | --- |
| **Select current backend ID** | If your application has EJB modules for which deployment code has been generated for multiple backend databases using an assembly tool, for **Select current backend ID**, specify the backend ID representing the backend database to be used when the EJB module runs.<br><br>This step is not shown if the **Deploy enterprise beans** setting is enabled on the **Select installation options** panel and if a database type other than `None` is specified on the **Provide options to perform the EJB Deploy** panel. |
| **Metadata for modules** | If your application has EJB 3.0 or Web 2.5 modules, you can lock deployment descriptors for one or more of the EJB 3.0 or Web 2.5 modules. If you set the `metadata-complete` attribute to `true` and lock deployment descriptors, the product writes the complete module deployment descriptor, including deployment information from annotations, to XML format. |
| **Provide options to perform the Web services deployment** | If the **Deploy Web services** setting is enabled on the **Select installation options** panel and your application uses Web services, then you can specify `wsdeploy` command options on the **Provide options to perform the Web services deployment** panel. For information on this panel, refer to descriptions of the `wsdeploy -cp` and `-jardir` options. |

5. On the Summary panel, verify the cell, node, and server onto which the application modules will install:
   a. Beside **Cell/Node/Server**, click **Click here**.
   b. Verify the settings.
   c. Return to the Summary panel.
   d. Click **Finish**.

## Results

Several messages are displayed, indicating whether your application file is installing successfully.

If **Validate input off/warn/fail** on the **Select installation options** panel is set to **warn**, the default, several validation warnings might be displayed. If the setting is **fail**, the validation warnings might cause errors.

If you receive an OutOfMemory error 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 error, 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 error, check the options you specified on the Java virtual machine page of the application server running the administrative console. You might increase the maximum heap size. Then, try installing the application file again.

## What to do next

After the application file installs successfully, do the following:

1. Save the changes to your configuration.

   The application is registered with the administrative configuration and application files are copied to the target directory, which is *app_server_root*/installedApps/*cell_name* by default or the directory that you designate.

   For a single-server product, application files are copied to the destination directory when the changes are saved.

   If you clicked the **Save** link in the application installation messages, the Preparing for the application installation panel displays again. Click **Applications** → **Application Types** → **WebSphere enterprise applications** to exit the panel and to see your application in the list of installed applications.

2. Start the application.
3. Test the application. For example, point a Web browser at the URL for the deployed application and examine the performance of the application. If necessary, edit the application configuration.

## Preparing for application installation settings

Use this page to specify an application or module to install.

To view this administrative console page, click **Applications** → **New application** → **New Enterprise Application**.

This page is the first Preparing for the application installation page. On this page, specify an application or module to install. You can install an enterprise application archive (EAR file), enterprise bean (EJB) module (JAR file), Session Initiation Protocol (SIP) module (SAR file), or Web module (WAR file).

The second Preparing for the application installation page has more installation options, such as to generate default bindings for incomplete existing bindings in your application or module.

***Path to the new application:***

Specifies the fully qualified path to the enterprise application file.

The file can be an .ear, .jar, .sar, or .war file.

During application installation, the product typically uploads application files from a client workstation running the browser to the server running the administrative console, and then deploys the application files on the server. In such cases, use the Web browser running the administrative console to select EAR, WAR, SAR, or JAR modules to upload to the server.

Use **Local file system** when the browser and application files are on the same computer.

Use **Remote file system** in the following situations:

- The application file resides on any node in the current cell context. Only .ear, .jar, .sar, or .war files are shown during the browsing.
- The application file resides on the file system of any of the nodes in a cell.
- The application file already resides on the computer running the application server. For example, the field value might be *profile_root*/installableApps/test.ear.

After the product transfers the application file, the **Remote file system** value shows the path of the temporary location on the server.

## Preparing for application installation binding settings

Use this page to select whether to view all installation options and to change the existing bindings for you application or module during installation. You can chose to generate default bindings for any incomplete bindings in the application or module or to assign specific bindings during installation.

This page is the second Preparing for the application installation page.

To view this administrative console page, click **Applications** → **New application** → **New Enterprise Application**, specify the path for the application or module to install, and then click **Next**.

The console page might not display all of the binding options listed in this topic. The contents of the application or module that you are installing determines which options are displayed on the console page. Also, the **Specify bindings to use** option displays only when updating an installed application.

***How do you want to install the application?:***

Specifies whether to show only installation options that require you to supply information or to show all installation options.

| Option | Description |
| --- | --- |
| **Fast Path - Prompt only when additional information is required** | Displays only those options that require your attention, based on the contents of your application or module. Use the fast path to install your application more easily because you do not need to examine all available installation options. |
| **Detailed - Show all installation options and parameters** | Displays all available installation options. |

*Specify bindings to use:*

Specifies whether to merge bindings when you update applications or to use new or existing bindings.

This setting is shown only when you update an installed application, and not when you install a new application.

| Option | Description |
| --- | --- |
| **Merge new and existing bindings** | Keeps the existing binding values of the installed application and adds new binding values in the updated application for incomplete bindings. Use merge if your updated application has binding values that differ from values specified for the installed application. The product assigns binding values in the following order:<br>1. Use existing binding values in the installed application.<br>2. If the installed application does not have a binding value, use the new binding value.<br>3. If both the installed application and the updated application do not have a binding value, use the default value. The product assigns a default value only if you select **Generate default bindings**. |
| **Use new bindings** | Uses binding values in the updated application. Does not use existing binding values in the installed application. |
| **Use existing bindings** | Uses existing binding values in the installed application. Does not use binding values in the updated application. |

*Generate default bindings:*

Specifies whether to generate default bindings and mappings. To view this setting, expand **Choose to generate default bindings and mappings**. If you select **Generate default bindings**, then the product completes any incomplete bindings in the application with default values. The product does not change existing bindings.

After you select **Generate default bindings**, you can advance directly to the Summary step and install the application if none of the steps have a red asterisk (*). 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.

**Note:** You do not need to specify Java Naming and Directory Interface (JNDI) values for EJB bean, local home, remote home, or business interfaces of EJB 3.0 modules. The product assigns container default values during run time. Similarly, for any EJB reference within an EJB 3.0, Web 2.4, or Web 2.5 module, you do not need to specify JNDI values because the product resolves the targets automatically during run time. Even when you select **Generate default bindings**, the product does not generate default values for those JNDI values but it does generate default values for other bindings such as virtual host.

If you select **Generate default bindings**, the product generates bindings as follows:

- Enterprise bean (EJB) JNDI names are generated in 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 or in its corresponding annotation for EJB 3.0 modules. The product does not generate default values for enterprise beans in an EJB 3.0 module because the run time provides container default values.
- EJB references are bound if an `<ejb-link>` is found. 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. The product does not generate default values for EJB reference in an EJB 3.0, Web 2.4, or Web 2.5 module because the run time provides container default values or automatically resolves the target references.
- Resource reference bindings are derived from the `<res-ref-name>` tag or its corresponding annotation for Java Platform, Enterprise Edition (Java EE) 5 modules. 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 and EJB 2.1 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 and EJB 2.1 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 action results in default data source settings for each JAR file. No bean-level data source bindings are generated.
- For EJB 2.0 or later message-driven beans deployed as Java EE Connector Architecture (JCA) 1.5-compliant resources, the JNDI names corresponding to activationSpec instances are generated in the form eis/*MDB_ejb-name*. Message destination references are bound if a `<message-destination-link>` is found, then the JNDI name is set to ejs/*message-destination-link*Name. Otherwise, the JNDI name is set to eis/*message-destination-ref*Name.
- For EJB 2.0 or later message-driven beans deployed against listener ports, the listener ports are derived from the message-driven bean `<ejb-name>` tag with the string `Port` appended.
- 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, if you experience errors, complete the following actions:

- Control the global JNDI names of one or more EJB files.
- Control data source bindings for container-managed persistence (CMP) beans. That is, you have multiple data sources and need more than one global data source.
- Map resource references to global resource JNDI names that are different from the java:comp/env name.

In such cases, you can change the behavior with an XML document, which is a custom strategy. Use the **Specific bindings file** setting to specify a custom strategy and see the setting description in this help file for examples.

***Override existing bindings:***

Specifies whether generated bindings are to replace existing bindings.

The default is to not override existing bindings. Select **Override existing bindings** to have generated bindings replace existing bindings.

***Specific bindings file:***

Specifies a bindings file that overrides the default binding.

Change the behavior of the default binding with an XML document, which is a custom strategy. Custom strategies extend the default strategy so you only need to customize those areas where the default strategy is insufficient. Thus, you only need to describe how you want to change the bindings generated by the default strategy; you do not have to define bindings for the entire application.

Use the following examples to override various aspects of the default bindings generator:

### Controlling an EJB JNDI name

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>helloEjb.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
         <ejb-name>HelloEjb</ejb-name>
         <jndi-name>com/acme/ejb/HelloHome</jndi-name>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

**Note:** Ensure that the setting for `<ejb-name>` matches the `ejb-name` entry in the EJB JAR deployment descriptor. Here the setting is `<ejb-name>HelloEjb</ejb-name>`.

### Setting the connection factory binding for an EJB JAR file

```
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>yourEjb20.jar</jar-name>
      <connection-factory>
        <jndi-name>eis/jdbc/YourData_CMP</jndi-name>
        <res-auth>Container</res-auth>
      </connection-factory>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

### Setting the connection factory binding for an EJB file

```
<?xml version="1.0">
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>yourEjb20.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourCmp20</ejb-name>
          <connection-factory>
           <jndi-name>eis/jdbc/YourData_CMP</jndi-name>
           <res-auth>PerConnFact</res-auth>
          </connection-factory>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
 </module-bindings>
</dfltbndngs>
```

**Note:** Ensure that the setting for `<ejb-name>` matches the `ejb-name` tag in the deployment descriptor. Here the setting is `<ejb-name>YourCmp20</ejb-name>`.

### Setting the message destination reference JNDI for a specific enterprise bean

This example shows an XML extract in a custom strategy file for setting message-destination-refs for a specific enterprise bean.

```
<?xml version="1.0">
 <!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
 <dfltbndngs>
  <module-bindings>
   <ejb-jar-binding>
    <jar-name>yourEjb21.jar</jar-name>
    <ejb-bindings>
     <ejb-binding>
      <ejb-name>YourSession21</ejb-name>
      <message-destination-ref-bindings>
       <message-destination-ref-binding>
        <message-destination-ref-name>jdbc/MyDataSrc</message-destination-ref-name>
        <jndi-name>eis/somAO</jndi-name>
       </message-destination-ref-binding>
      </message-destination-ref-bindings>
     </ejb-binding>
    </ejb-bindings>
   </ejb-jar-binding>
  </module-bindings>
 </dfltbndngs>
```

**Note:** Ensure that the setting for `<ejb-name>` matches the `ejb-name` tag in the deployment descriptor. Here the setting is `<ejb-name>YourSession21</ejb-name>`. Also ensure that the setting for `<message-destination-ref-name>` matches the `message-destination-ref-name` tag in the deployment descriptor. Here the setting is `<message-destination-ref-name>jdbc/MyDataSrc</message-destination-ref-name>`.

**Overriding a resource reference binding from a WAR, EJB JAR file, or Java EE client JAR file**

This example shows code for overriding a resource reference binding from a WAR file. Use similar code to override a resource reference binding from an enterprise bean (EJB) JAR file or a Java EE client JAR file.

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <war-binding>
      <jar-name>hello.war</jar-name>
      <resource-ref-bindings>
        <resource-ref-binding>
          <resource-ref-name>jdbc/MyDataSrc</resource-ref-name>
          <jndi-name>war/override/dataSource</jndi-name>
        </resource-ref-binding>
      </resource-ref-bindings>
    </war-binding>
  </module-bindings>
</dfltbndngs>
```

**Note:** Ensure that the setting for `<resource-ref-name>` matches the `resource-ref` tag in the deployment descriptor. In the previous example, the setting is `<resource-ref-name>jdbc/MyDataSrc</resource-ref-name>`.

**Overriding the JNDI name for a message-driven bean deployed as a JCA 1.5-compliant resource**

This example shows an XML extract in a custom strategy file for overriding the Java Message Service (JMS) activationSpec JNDI name for an EJB 2.0 or later message-driven bean deployed as a JCA 1.5-compliant resource.

```
<?xml version="1.0"?>
 <!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
  <dfltbndngs>
```

```
  <module-bindings>
   <ejb-jar-binding>
     <jar-name>YourEjbJar.jar</jar-name>
     <ejb-bindings>
      <ejb-binding>
        <ejb-name>YourMDB</ejb-name>
        <activationspec-jndi-name>activationSpecJNDI</activationspec-jndi-name>
      </ejb-binding>
     </ejb-bindings>
   </ejb-jar-binding>
  </module-bindings>
 </dfltbndngs>
```

#### Overriding the JMS listener port name for an EJB 2.0, 2.1, or 3.0 message-driven bean

This example shows an XML extract in a custom strategy file for overriding the JMS listener port name for an EJB 2.0 or later message-driven bean deployed against a listener port.

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>YourEjbJar.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourMDB</ejb-name>
          <listener-port>yourMdbListPort</listener-port>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

#### Overriding an EJB reference binding from an EJB JAR, WAR file, or EJB file

This example shows code for overriding an EJB reference binding from an EJB JAR file. Use similar code to override an EJB reference binding from a WAR file or an EJB file.

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>YourEjbJar.jar</jar-name>
      <ejb-ref-bindings>
        <ejb-ref-binding>
          <ejb-ref-name>YourEjb</ejb-ref-name>
          <jndi-name>YourEjb/JNDI</jndi-name>
        </ejb-ref-binding>
      </ejb-ref-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

***Specify unique prefix for beans:***

Specifies a string that the product applies to the beginning of generated enterprise bean JNDI names. The prefix must be unique within the cell or node.

The default is to not specify a unique prefix for beans.

***Default bindings for EJB 1.1 CMP beans:***

Specifies the default data source JNDI name and other bindings for container-managed persistence (CMP) 1.1 beans.

The default is to not use default bindings for EJB 1.1 CMP beans.

If you select **Default bindings for EJB 1.1 CMP beans**, specify the JNDI name for the default data source to be used with the CMP 1.1 beans. Also specify the user name and password for this default data source.

***Default connection factory bindings:***

Specifies the default connection factory JNDI name.

The default is to not use default connection factory bindings. Select **Default connection factory bindings** to specify bindings for connection factories.

If you select **Default connection factory bindings**, specify the JNDI name for the default connection factory to be used. Also specify whether the resource authorization is for the application or container-wide.

***Use default virtual host name for Web and SIP modules:***

Specifies the virtual host for the Web module (WAR file) or Session Initiation Protocol (SIP) module (SAR file).

The default is to not use default virtual host name for Web or SIP modules. If you select **Use default virtual host name for Web and SIP modules**, specify a default host name.

## Select installation options settings

Use this panel to specify options for the installation of a Java Platform, Enterprise Edition (Java EE) application onto a WebSphere Application Server deployment target. Default values for the options are used if you do not specify a value. After application installation, you can specify values for many of these options from an enterprise application settings page.

To view this administrative console panel, click **Applications** → **New application** → **New Enterprise Application** and then specify values as needed for your application on the Preparing for application installation pages.

The Select installation options panel is the same for the application installation and update wizards.

***Precompile JavaServer Pages files:***

Specify whether to precompile JavaServer Pages (JSP) files as a part of installation. The default is not to precompile JSP files.

For this option, install only onto a Version 6.1 or later deployment target.

If you select **Precompile JavaServer Pages files** and try installing your application onto an earlier deployment target such as Version 5.x, the installation is rejected. You can deploy applications to only those deployment targets that have same version as the product. If applications are targeted to servers that have an earlier version than the product, then you cannot deploy to those targets.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

***Directory to install application:***

Specifies the directory to which the enterprise archive (EAR) file will be installed.

By default, the EAR file is installed in the *profile_root*/installedApps/*cell_name*/*application_name*.ear directory.

Setting options include the following:

- Do not specify a value and leave the field empty.

  The default value is ${APP_INSTALL_ROOT}/*cell_name*, where the ${APP_INSTALL_ROOT} variable is *profile_root*/installedApps. A directory having the EAR file name of the application being installed is appended to ${APP_INSTALL_ROOT}/*cell_name*. Thus, if you do not specify a directory, the EAR file is installed in the *profile_root*/installedApps/*cell_name*/*application_name*.ear directory.
- Specify a directory.

  If you specify a directory for **Directory to install application**, the application is installed in *specified_path*/*application_name*.ear directory. A directory having the EAR file name of the application being installed is appended to the path that you specify for **Directory to install application**. For example, if you are installing Clock.ear and specify `C:/myapps` on Windows machines, the application is installed in the myapps/Clock.ear directory. The ${APP_INSTALL_ROOT} variable is set to the specified path.
- Specify `${APP_INSTALL_ROOT}/${CELL}` for the initial installation of the application.

  If you intend to export the application from one cell and later install the exported application on a different cell, specify the `${CELL}` variable for the initial installation of the application. For example, specify `${APP_INSTALL_ROOT}/${CELL}` for this setting. Exporting the application creates an enhanced EAR file that has the application and its deployment configuration. The deployment configuration retains the cell name of the initial installation in the destination directory unless you specify the `${CELL}` variable. Specifying the `${CELL}` variable ensures that the destination directory has the current cell name, and not the original cell name.

  **Note:** If an installation directory is not specified when an application is installed on a single-server configuration, the application is installed in ${APP_INSTALL_ROOT}/*cell_name*. When the server is made a part of a multiple-server configuration (using the addNode utility), the cell name of the new configuration becomes the cell name of the deployment manager node. If the `-includeapps` option is used for the addNode utility, then the applications that are installed prior to the addNode operation still use the installation directory ${APP_INSTALL_ROOT}/*cell_name*. However, an application that is installed after the server is added to the network configuration uses the default installation directory ${APP_INSTALL_ROOT}/*network_cell_name*. To move the application to the ${APP_INSTALL_ROOT}/*network_cell_name* location upon running the addNode operation, explicitly specify the installation directory as `${APP_INSTALL_ROOT}/${CELL}` during installation. In such a case, the application files can always be found under ${APP_INSTALL_ROOT}/*current_cell_name*.
- If the application has been exported and you are installing the exported EAR file in a different cell or location, specify ${APP_INSTALL_ROOT}/*cell_name*/*application_name*.ear if you did not specify `${APP_INSTALL_ROOT}/${CELL}` for the initial installation.

  The exported EAR file is an enhanced EAR file that has the application and its deployment configuration. The deployment configuration retains the value for **Directory to install application** that was used for the previous installation of the application. Unless you specify a different value for **Directory to install application** for this installation, the enhanced EAR file will be installed to the same directory as for the previous installation.

  If you did not specify the `${CELL}` variable during the initial installation, the deployment configuration uses the cell name of the initial installation in the destination directory. If you are installing on a different cell, specify ${APP_INSTALL_ROOT}/*cell_name*/*application_name*.ear, where *cell_name* is the name of the cell to which you want to install the enhanced EAR file. If you do not designate the current cell name, *cell_name* will be the original cell name even though you are installing the enhanced EAR file on a cell that has a different name.

- Specify an absolute path or a use pathmap variable.

  You can specify an absolute path or use a pathmap variable such as `${MY_APPS}`. You can use a pathmap variable in any installation.

This **Directory to install application** field is the same as the **Location (full path)** setting on an Application binaries page.

| | |
|---|---|
| **Data type** | String |
| **Units** | Full path name |

***Distribute application:***

Specifies whether the product expands application binaries in the installation location during installation and deletes application binaries during uninstallation. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified.

On single-server products, the binaries are deleted when you uninstall and save changes to the configuration.

On multiple-server products, the binaries are deleted when you uninstall and save changes to the configuration and synchronize changes.

If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

**Note:** If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you.

This **Distribute application** field is the same as the **Enable binary distribution, expansion and cleanup post uninstallation** setting on an Application binaries page.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | true |

***Use binary configuration:***

Specifies whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the deployment.xml file (default), or those located in the enterprise archive (EAR) file. Select this setting for applications installed on Version 6.0 or later deployment targets only. This setting is not valid for applications installed on 5.x deployment targets.

The default (false) is not to use the binding, extensions, and deployment descriptors located in deployment.xml. To use the binding, extensions, and deployment descriptors located in the EAR file, enable this setting (true).

This **Use binary configuration** field is the same as the **Use configuration information in binary** setting on an Application binaries page.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

***Deploy enterprise beans:***

Specifies whether the EJBDeploy tool runs during application installation.

The tool generates code needed to run enterprise bean (EJB) files. You must enable this setting in the following situations:
- The EAR file was assembled using an assembly tool such as Rational Application Developer and the EJBDeploy tool was not run during assembly.
- The EAR file was not assembled using an assembly tool such as Rational Application Developer.
- The EAR file was assembled using versions of the Application Assembly Tool (AAT) previous to Version 5.0.

The EJB deployment tool runs during installation of EJB 1.x or 2.x modules. The EJB deployment tool does not run during installation of EJB 3.0 modules.

For this option, install only onto a Version 6.1 or later deployment target.

If you select **Deploy enterprise beans** and try installing your application onto an earlier deployment target such as Version 6.0, the installation is rejected. You can deploy applications to only those targets that have same WebSphere version as the product. If applications are targeted to servers that have an earlier version than the product, then you cannot deploy to those targets.

Also, if you select **Deploy enterprise beans** and specify a database type on the **Provide options to perform the EJB Deploy** panel, previously defined backend IDs for all of the EJB modules are overwritten by the chosen database type. To enable backend IDs for individual EJB modules, set the database type to `""` (null) on the **Provide options to perform the EJB Deploy** panel.

Enabling this setting might cause the installation program to run for several minutes.

**Data type** Boolean
**Default** true (`false` for EJB 3.0 modules)

### *Application name:*

Specifies a logical name for the application. An application name must be unique within a cell and cannot contain an unsupported character.

An application name cannot begin with a period (`.`), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

| **Unsupported characters** | | |
|---|---|---|
| **/** forward slash | **$** dollar sign | **'** single quote mark |
| **** backslash | **=** equal sign | **"** double quote mark |
| ***** asterisk | **%** percent sign | **\|** vertical bar |
| **,** comma | **+** plus sign | **<** left angle bracket |
| **:** colon | **@** at sign | **>** right angle bracket |
| **;** semi-colon | **#** hash mark | **&** ampersand (and sign) |
| **?** question mark | **]]>** No specific name exists for this character combination | |

This **Application name** field is the same as the **Name** setting on an Enterprise application settings page.

**Data type** String

### *Create MBeans for resources:*

Specifies whether to create MBeans for resources such as servlets or JSP files within an application when the application starts. The default is to create MBeans.

This field is the same as the **Create MBeans for resources** setting on a Startup behavior page.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | true |

*Override class reloading settings for Web and EJB modules:*

Specifies whether the product run time detects changes to application classes when the application is running. If this setting is enabled and if application classes are changed, then the application is stopped and restarted to reload updated classes.

The default is not to enable class reloading.

This field is the same as the **Override class reloading settings for Web and EJB modules** setting on an Class loading and update detection page.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

*Reload interval in seconds:*

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

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

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

This **Reload interval in seconds** field is the same as the **Polling interval for updated files** setting on a Class loading and update detection page.

| | |
|---|---|
| **Data type** | Integer |
| **Units** | Seconds |
| **Default** | 3 |

*Deploy Web services:*

Specifies whether the Web services deploy tool `wsdeploy` runs during application installation.

The tool generates code needed to run applications using Web services. The default is not to run the wsdeploy tool. You must enable this setting if the EAR file contains modules using Web services and has not previously had the wsdeploy tool run on it, either from the **Deploy** menu choice of an assembly tool or from a command line.

For this option, install only onto a Version 6.1 or later deployment target.

If you select **Deploy Web services** and try installing your application onto an earlier deployment target such as Version 5.x, the installation is rejected. You can deploy applications to only those targets that have same version as the product. If applications are targeted to servers that have an earlier version than the

product, then you cannot deploy to those targets.

**Data type** Boolean
**Default** false

***Validate input off/warn/fail:***

Specifies whether the product examines the application references specified during application installation or updating and, if validation is enabled, warns you of incorrect references or fails the operation.

An application typically refers to resources using data sources for container managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the application is defined in the scope of the deployment target of that application.

Select **off** for no resource validation, **warn** for warning messages about incorrect resource references, or **fail** to stop operations that fail as a result of incorrect resource references.

This **Validate input off/warn/fail** field is the same as the **Application reference validation** setting on an Enterprise application settings page.

**Data type** String
**Default** warn

***Process embedded configuration:***

Specifies whether the embedded configuration should be processed. An embedded configuration consists of files such as resource.xml and variables.xml. When selected or `true`, the embedded configuration is loaded to the application scope from the .ear file. If the .ear file does not contain an embedded configuration, the default is `false`. If the .ear file contains an embedded configuration, the default is `true`.

**Data type** Boolean
**Default** false

***File permission:***

Specifies access permissions for application binaries for installed applications that are expanded to the directory specified.

The **Distribute application** option must be enabled to specify file permissions.

You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the multiple-selection list. List selections overwrite file permissions set in the text field.

You can set one or more of the following file permission strings in the list. Selecting multiple options combines the file permission strings.

| Multiple-selection list option | File permission string set |
| --- | --- |
| **Allow all files to be read but not written to** | `.*=755` |
| **Allow executables to execute** | `.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755` |
| **Allow HTML and image files to be read by everyone** | `.*\.htm=755#.*\.html=755#.*\.gif=755#.*\.jpg=755` |

Instead of using the multiple-selection list to specify file permissions, you can specify a file permission string in the text field. File permissions use a string that has the following format:

```
file_name_pattern=permission#file_name_pattern=permission
```

where *file_name_pattern* is a regular expression file name filter (for example, `.*\\.jsp` for all JSP files), *permission* provides the file access control lists (ACLs), and `#` is the separator between multiple entries of *file_name_pattern* and *permission*. If `#` is a character in a *file_name_pattern* string, use `\#` instead.

If multiple file name patterns and file permissions in the string match a uniform resource identifier (URI) within the application, then the product uses the most stringent applicable file permission for the file. For example, if the file permission string is `.*\\.jsp=775#a.*\\.jsp=754`, then the abc.jsp file has file permission 754.

**Note:** Using regular expressions for file matching pattern compares an entire string URI against the specified file permission pattern. You must provide more precise matching patterns using regular expressions as defined by Java programming API. For example, suppose the following directory and file URIs are processed during a file permission operation:

| | |
|---|---|
| 1 | /opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war |
| 2 | /opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp |
| 3 | /opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF/MANIFEST.MF |
| 4 | /opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/WEB-INF/classes/MyClass.class |
| 5 | /opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/mydir/MyClass2.class |
| 6 | /opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF |

The file pattern matching results are:
- `MyWarModule.war` does not match any of the URIs
- `.*MyWarModule.war.*` matches all URIs
- `.*MyWarModule.war$` matches only URI 1
- `.*\\.jsp=755` matches only URI 2
- `.*META-INF.*` matches URIs 3 and 6
- `.*MyWarModule.war/.*/.*\.class` matches URIs 4 and 5

If you specify a directory name pattern for **File permissions**, then the directory permission is set based on the value specified. Otherwise, the **File permissions** value set on the directory is the same as its parent. For example, suppose you have the following file and directory structure:

```
/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
```

and you specify the following file pattern string:

```
.*MyApp.ear$=755#.*\.jsp=644
```

The file pattern matching results are:
- Directory MyApp.ear is set to 755
- Directory MyWarModule.war is set to 755
- Directory MyWarModule.war is set to 755

**Note:** Regardless of the operation system, always use a forward slash (/) as a file path separator in file patterns.

Access permissions specified here are at the application level. You can also specify access permissions for application binaries in the node-level configuration. The node-level file permissions specify the maximum (most lenient) permissions that can be given to application binaries. Access permissions specified here at application level can only be the same as or more restrictive than those specified at the node level.

This setting is the same as the **File permissions** field on the Application binaries page.

| | |
|---|---|
| **Data type** | String |

***Application build identifier:***

Specifies an uneditable string that identifies the build version of the application.

This **Application build identifier** field is the same as the **Application build level** field on the Application binaries page.

| | |
|---|---|
| **Data type** | String |

***Business-level application name:***

Specifies whether the product creates a new business-level application with the enterprise application that you are installing or makes the enterprise application a composition unit of an existing business-level application.

The default is to create a new business-level application with a setting value of `WebSphere:blaname=Anyasset,blaedition=BASE`. When you select to create a new business-level application from the drop-down list, the product creates a business-level application that has the same name as your enterprise application.

To add your enterprise application to an existing business-level application, select an existing business-level application from the drop-down list. The product makes your enterprise application a composition unit of the existing business-level application.

| | |
|---|---|
| **Data type** | String |
| **Default** | Create a new business-level application that has the same name as the enterprise application that you are installing.<br><br>`WebSphere:blaname=Anyasset,blaedition=BASE` |

***Asynchronous request dispatch type:***

Specifies whether Web modules can dispatch requests concurrently on separate threads and, if so, whether the server or client dispatches the requests. Concurrent dispatching can improve servlet response time.

If operations are dependant on each other, do not enable asynchronous request dispatching. Select **Disabled**. Concurrent dispatching might result in errors when operations are dependant.

Select **Server side** to enable the server to dispatch requests concurrently. Select **Client side** to enable the client to dispatch requests concurrently.

| | |
|---|---|
| **Data type** | String |
| **Default** | Disabled |

***Allow EJB reference targets to resolve automatically:***

Specifies whether the product assigns default JNDI values for or automatically resolves incomplete EJB reference targets.

Select this option to enable EJB reference targets to resolve automatically if the references are from EJB 2.1 or earlier modules or from Web 2.3 or earlier modules. If you enable this option, the runtime container provides a default value or automatically resolves the EJB reference for any EJB reference that does not have a binding.

If you selected **Generate default bindings** on the Preparing for application installation page, then you do not need to select this option. The product generates default values.

If you select **Allow EJB reference targets to resolve automatically**, all modules in the application must share one deployment target. If you select this option and all of the application modules do not share a common server, after you click **Finish** on the Summary page, the product displays a warning message and does not install the application. You must deselect this setting before you click **Finish** to install the application.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

## Provide options to perform the EJB Deploy settings

Use this panel to specify options for the enterprise bean (EJB) deployment tool. The tool generates code needed to run enterprise bean files. You can specify extra class paths, Remote Method Invocation compiler (RMIC) options, database types, and database schema names to be used while running the EJB deployment tool.

This administrative console panel is a step in the application installation and update wizards. To view this panel, you must select **Deploy enterprise beans** on the **Select installation options** panel. Thus, to view this panel, click **Applications** → **New Application** → **New Enterprise Application** → ***application_path*** → **Next** → **Detailed - Show all installation options and parameters** → **Next** → **Deploy enterprise beans** → **Next** → **Step: Provide options to perform the EJB Deploy**.

You can specify the EJB deployment tool options on this panel when installing or updating an application that contains EJB modules. The EJB deployment tool runs during installation of EJB 1.x or 2.x modules. The EJB deployment tool does not run during installation of EJB 3.0 modules.

The options that you specify set parameter values for the ejbdeploy command. The tool, and thus the ejbdeploy command, is run on the enterprise archive (EAR) file during installation after you click **Finish** on the **Summary** panel of the wizard.

***Class path:***

Specifies the class path of one or more zipped or Java archive (JAR) files on which the JAR or EAR file being installed depends.

To specify the class paths of multiple zipped and JAR files, the zipped and JAR file names must be fully qualified, separated by semicolons, and enclosed in double quotation marks. For example:

```
path\myJar1.jar;path\myJar2.jar;path\myJar3.jar
```

**Class path** is the same as the ejbdeploy command parameter `-cp` *class_path*.

| | |
|---|---|
| **Data type** | String |
| **Default** | null |

*RMIC:*

Specifies whether the EJB deployment tool passes RMIC options to the Remote Method Invocation compiler. Refer to RMI Tools documentation for information on the options.

Separate options by a space and enclose them in double quotation marks. For example:

```
"-nowarn -verbose"
```

The **RMIC** setting is the same as the ejbdeploy command parameter `-rmic "options"`.

| | |
|---|---|
| **Data type** | String |
| **Default** | null |

*Database type:*

Specifies the name of the database vendor, which is used to determine database column types, mapping information, Table.sql, and other information. Select a database type or the empty choice from the drop-down list. The list contains the names of valid database vendors. Selecting the empty choice sets the database type to "" (null).

If you specify a database type, previously defined backend IDs for all of the EJB modules are overwritten by the chosen database type. To enable backend IDs for individual EJB modules, select the empty choice to set the database type to null.

The backend IDs SQL92 (1992 SQL Standard) and SQL99 (1999 SQL Standard) are deprecated. Although the SQL92 and SQL99 backend IDs are available in the list, they are deprecated.

**Database type** is the same as the ejbdeploy command parameter `-dbvendor name`.

| | |
|---|---|
| **Data type** | String |
| **Default** | DB2UDB_V82 |

*Database schema:*

Specifies the name of the schema that you want to create.

The EJB deployment tool saves database information in the schema document in the JAR or EAR file, which means that the options do not need to be specified again. It also means that when a JAR or EAR is generated, the correct database must be defined at that point because it cannot be changed later.

If the name of the schema contains any spaces, the entire name must be enclosed in double quotes. For example:

```
"my schema"
```

**Database schema** is the same as the ejbdeploy command parameter `-dbschema "name"`.

| | |
|---|---|
| **Data type** | String |
| **Default** | null |

*Database access type:*

Specifies the database access type for a DB2 database that supports Structured Query Language for Java (SQLJ). Use SQLJ to develop data access applications that connect to DB2 databases. SQLJ is a set of

programming extensions that support use of the Java programming language to embed statements that provide SQL (Structured Query Language) database requests.

To view this setting, you must select a DB2 backend database that supports SQLJ from the **Database type** drop-down list.

Available database access types include `JDBC` and `SQLJ`.

| | |
|---|---|
| **Data type** | String |
| **Default** | JDBC |

***SQLJ class path:***

Specifies the class path of the DB2 SQLJ tool sqlj.zip file. The product uses this class path to run the DB2 SQLJ tool during application installation and generate SQLJ profiles (.ser files).

To view this setting, you must select a DB2 backend database that supports SQLJ from the **Database type** drop-down list.

When you reinstall an application EAR file, the product deletes any existing SQLJ profiles and creates new profiles.

If you do not specify a class path, the product displays a warning about the missing class path. After you specify a valid class path, you can continue using the wizard for the application installation.

You can customize or add bindings to the generated SQLJ profile after the product installs the application. Use the administrative console SQLJ profiles and pureQuery bind files page accessed by clicking **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **SQLJ profiles and pureQuery bind files**.

| | |
|---|---|
| **Data type** | String |
| **Default** | null |

***JDK compliance level:***

Specifies the Java developer kit compiler compliance level as *1.4*, *5.0*, or *6.0* when you include application source files for compilation.

The default is to use whatever developer kit version the ejbdeploy command is using. If your application is using new functionality defined in Version 5.0 or 6.0 or you are including source files (which is not recommended), then you must specify the Version 5.0 or 6.0 level.

**JDK compliance level** is the same as the ejbdeploy command parameter `-complianceLevel "1.4" | "5.0" | "6.0"`.

| | |
|---|---|
| **Data type** | String |
| **Default** | null (empty string) |

## Bind listeners for message-driven beans settings

Use this panel to specify bindings for message-driven beans in your application or module.

To view this administrative console panel, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Message Driven Bean listener bindings**. This panel is the same as the **Bind listeners for message-driven beans** panel on the application installation and update wizards.

Each message-driven bean must be bound to a listener port name or to an activation specification Java Naming and Directory Interface (JNDI) name.

Provide a listener port name if your application uses any of the following Java Message Service (JMS) providers:
- Version 5 default messaging
- WebSphere MQ
- Generic

Provide an activation specification JNDI name if your application's resources are configured using the default messaging provider or any generic J2C resource adapter that supports inbound messaging.

Not providing valid listener port names or activation specification JNDI names results in the following errors:
- If neither a listener port name or an activation specification JNDI name is specified for a message driven bean, then a validation error is displayed after you click **Finish** on the **Summary** panel.
- If the module containing the message-driven bean is deployed on a 5.x deployment target and a listener port is not specified, then a validation error is displayed after you click **Next**.
- If multiple message driven beans are linked to the same destination, specify the same destination JNDI name for each message driven bean. If you specify different destination JNDI names, a validation error is displayed and all JNDI specifications after the first one are ignored.

To apply binding changes to multiple mappings:
1. In the list of mappings, select the **Select** check box beside each EJB module that you want mapped to a particular binding.
2. Expand **Apply Multiple Mappings**.
3. Specify a listener port name or select a target resource JNDI name for an activation specification.
4. If you are defining a binding for an activation specification, optionally specify the following:

   **Destination JNDI name**
   : For resource adapters that support JMS, specify javax.jms.Destinations so the resource adapter can service messages from the JMS destination. A destination JNDI name set as part of application deployment take precedence over properties set on an activation specification administrative object.

   **Target resource JNDI Name**
   : Specify the target resource JNDI name when mapping a message-driven bean to an activation specification.

   **ActivationSpec authentication alias**
   : Specify an authentication alias that is used to access the user name and password that are set on the configured J2C activation specification. Authentication alias properties set as part of application deployment take precedence over properties set on an activation specification administrative object.
5. Click **Apply**.
6. Click **OK** or **Next**.

***EJB module:***

Specifies the name of the module that contains the enterprise bean.

***EJB:***

Specifies name of an enterprise bean in the application.

***URI:***

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

***Messaging type:***

Specifies the type of message-driven bean.

***Bindings:***

Specifies a listener port name or an activation specification JNDI name for the message-driven bean. When a message-driven enterprise bean is bound to an activation specification JNDI name you can also specify the destination JNDI name and the authentication alias.

Bindings specify JNDI names for the referenceable and referenced artifacts in an application. An example JNDI name for a listener port to be used by a `Store` application might be `StoreMdbListener`. The binding definition is stored in IBM bindings files such as ibm-ejb-jar-bnd.xmi.

## Example: Installing an EAR file using the default bindings

If application bindings were not specified for all enterprise beans or resources in an enterprise application during application development or assembly, you can select to generate default bindings. After application installation, you can modify the bindings as needed using the administrative console.

### Before you begin

This topic assumes that the application can run on a Web server.

### About this task

This topic describes how to install a simple .ear file using the default bindings. You can follow the steps to install any application, including applications provided with the product in the samples or installableApps subdirectory.

1. Click **Applications** → **New application** → **New Enterprise Application** in the console navigation tree.
2. On the first Preparing for application install page, specify the full path name of the EAR file.
   a. For **Path to the new application**, specify the full path name of the .ear file. For this example, the base file name is my_appl.ear and the file resides on a server at C:\sample_apps.

      For this example, the base file name is my_appl.ear and the file resides on a server at /home/myuserid/myapps. Thus, enter the fully qualified path name for the file, /home/myuserid/myapps/my_appl.ear.

   b. Click **Next**.
3. On the second Preparing for application install page, choose to generate default bindings.
   a. Expand **Choose to generate default bindings and mappings**.
   b. Select **Generate default bindings**.

      Using the default bindings causes any incomplete bindings in the application to be filled in with default values. the product does not change existing bindings. By choosing this option, you can skip many of the steps of the application installation wizard and go directly to the Summary step.

   c. Click **Next**.
4. If application security warnings are displayed, read the warnings and click **Continue**.
5. On the Install New Application page, click the step number for **Map modules to servers**, and verify the cell, node, and server onto which the application files will install.

a. From the **Clusters and servers** list, select the server onto which the application files will install.
b. Select all of the application modules.
c. Click **Next**.

On the **Map modules to servers** panel, you can map modules to other servers such as Web servers. If you want a Web server to serve the application, use the **Ctrl** key to select an application server or cluster and the Web server together in order to have the plug-in configuration file plugin-cfg.xml for that Web server generated based on the applications which are routed through it.

6. On the Install New Application page, click the step number beside **Summary**, the last step.
7. On the Summary panel, click **Finish**.

## What to do next

Examine the application installation progress messages. If the application installs successfully, save your administrative configuration. You can now see the name of your application in the list of deployed applications on the Enterprise applications page accessed by clicking **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree.

If the application does not install successfully, read the messages to identify why the installation failed. Correct problems with the application as needed and try installing the application again.

If the application has a Web module, try opening a browser on the application.

1. Point a Web browser at the URL for the deployed application.

   The URL typically has the format `http://`*`host_name`*`:9060/`*`Web_module_name`*, where *host_name* is your valid Web server and 9060 is the default port number.
2. Examine the performance of the application.

If the application does not perform as desired, edit the application configuration, then save and test it again.

# Example: Installing a Web Services Sample with the console

The product provides a Web Services sample application that you can install on a Version 7.x application server.

## Before you begin

During installation, select to install the sample applications. After installation, ensure that your product installation has a Version 7.x application server onto which you can install the Web Services Sample.

## About this task

Installing the sample applications adds the JaxWSServicesSamples.ear enterprise application and supporting Java archives (JAR files) to the *app_server_root*/samples/lib/JaxWSServicesSamples directory of your product installation.

This topic describes how to install and start the JaxWSServicesSamples.ear enterprise application using an administrative console.

1. Click **Applications** → **New application** → **New Enterprise Application** in the console navigation tree.
2. On the first Preparing for application installation page, specify to install JaxWSServicesSamples.ear.
   a. Click **Local file system** or **Remote file system** and specify the full path name of the JaxWSServicesSamples.ear file.

   *app_server_root*/samples/lib/JaxWSServicesSamples/JaxWSServicesSamples.ear

   b. Click **Next**.

3. On the second Preparing for application installation page, select the fast path option.
   a. Select **Fast Path - Prompt only when additional information is required**.
   b. Click **Next**.
4. Click **Next** on each panel until you reach the Summary panel.

   Do not go directly from Step 1 to the Summary panel. You must click **Next** on each panel that has mandatory settings to enter values for those settings. Simply click **Next** to enter the default values. You optionally can change the values to suit your environment.
5. On the Summary panel, verify the cell, node, and server onto which the application modules will install, and then click **Finish**.
6. Examine the application installation progress messages.

   If the application installs successfully, the message *Application JaxWSServicesSamples installed successfully* is displayed. Click **Save**. After the configuration changes are saved, you can see the name of the application in the list of deployed applications on the Enterprise applications page accessed by clicking **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree.

   If the application does not install successfully, read the messages to identify why the installation failed. Correct problems with the server or application and try installing the application again.

## Results

The **JaxWSServicesSamples** application is in the list of deployed applications on the Enterprise applications page.

## What to do next

After the application installs successfully, do the following:
1. Start the application.

   On the Enterprise applications page, select the check boxes beside **JaxWSServicesSamples**, and then click **Start**.
2. Test the application. Point your Web browser at:

   `http://localhost:9080/wssamplesei/demo`

   If the localhost address does not load, substitute the host name (IP address) of the computer for localhost; for example, http://9.22.33.44:9080/wssamplesei/demo.

   If you have another WebSphere Application Server installation on your machine, the server port number is likely not 9080. See the Ports table in the administrative console to find the `WC_defaulthost` server port number. Click **Servers** → **Server Types** → **WebSphere application servers** → ***server1*** → **Ports**. The Ports table lists the important ports:

| Port name | Description |
|---|---|
| `WC_adminhost` | Port used to open an unsecure administrative console in the URL `http://`*host_name*`:`*administrative_port*`/ibm/console` |
| `WC_adminhost_secure` | Port used to open a secure administrative console in the URL `http://`*host_name*`:`*administrative_port*`/ibm/console` |
| `WC_defaulthost` | Port used to test running applications in the URL `http://`*host_name*`:`*server_port*`/`*servlet_name* |
| `WC_defaulthost_secure` | Port used to securely test running applications in the URL `http://`*host_name*`:`*server_port*`/`*servlet_name* |

You can also view the running Web Services Sample in a Web browser open on the Samples Gallery at http://localhost:9080/WSsamples. As needed, substitute the host name (IP address) of the computer for localhost and the port name for 9080. After installation of the Web Services Sample, click **Installed**

**Samples** → **Web Services Samples** → **JAX-WS Web Services Samples**.

## Installing enterprise modules with JSR-88

You can install Java Platform, Enterprise Edition (Java EE) modules on an application server provided by a WebSphere Application Server product using the Java EE Application Deployment API specification (JSR-88).

### Before you begin

JSR-88 defines standard application programming interfaces (APIs) to enable deployment of Java EE applications and stand-alone modules to Java EE product platforms. The Java EE Application Deployment specification Version 1.1 is available at http://java.sun.com/j2ee/tools/deployment/reference/docs/index.html as part of the Java 2 Platform, Enterprise Edition (J2EE) 1.4 Application Server Developer Release.

Read about JSR-88 and APIs used to manage applications at http://java.sun.com/j2ee/tools/deployment/.

### About this task

JSR-88 defines a contract between a tool provider and a platform that enables tools from multiple vendors to configure, deploy and manage applications on any Java EE product platform. The tool provider typically supplies software tools and an integrated development environment (IDE) for developing and assembly of Java EE application modules. The Java EE platform provides application management functions that deploy, undeploy, start, stop, and otherwise manage Java EE applications.

WebSphere Application Server is a Java EE specification-compliant platform that implements the JSR-88 APIs. Complete the following steps to deploy (install) Java EE modules on an application server provided by the WebSphere Application Server platform.

1. Code a Java program that can access the JSR-88 DeploymentManager class for the product.
   a. Write code that finds the JAR manifest attribute J2EE-DeploymentFactory-Implementation-Class.

      Under JSR-88, your code finds the DeploymentFactory using the JAR manifest attribute J2EE-DeploymentFactory-Implementation-Class. The following product application management JAR files contain this attribute and provide support:

| Environment | JAR file containing the manifest attribute |
|---|---|
| Application server | *app_server_root*/plugins/com.ibm.ws.admin.services.jar |
| Application client | *app_client_root*/plugins/com.ibm.ws.j2ee.client.jar |
| Thin application client | *app_client_root*/runtimes/com.ibm.ws.admin.client_7.0.0.jar |

      After your code finds the DeploymentFactory, the deployment tool can create an instance of the WebSphere DeploymentFactory and register the instance with its DeploymentFactoryManager.

      Example code for the application server environment follows. The example code requires that you use the development kit shipped with the product or use the pluggable client for deployment of stand-alone modules. See *WebSphere Application Server detailed system requirements* at http://www.ibm.com/support/docview.wss?rs=180&uid=swg27006921 for information on supported development kits.

```
import javax.enterprise.deploy.shared.factories.DeploymentFactoryManager;
import javax.enterprise.deploy.spi.DeploymentManager;
import javax.enterprise.deploy.spi.factories.DeploymentFactory;
import java.util.jar.JarFile;
import java.util.jar.Manifest;

// Get the DeploymentFactory implementation class from the MANIFEST.MF file.
File jsr88Jar = new File(wasHome + "/plugins/com.ibm.ws.admin.services.jar");
JarFile jarFile = new JarFile(jsr88Jar);
Manifest manifest = jarFile.getManifest();
```

```
Attributes attributes = manifest.getMainAttributes();
String key = "J2EE-DeploymentFactory-Implementation-Class";
String className = attributes.getValue(key);
// Get an instance of the DeploymentFactoryManager
DeploymentFactoryManager dfm = DeploymentFactoryManager.getInstance();

// Create an instance of the WebSphere Application Server DeploymentFactory.
Class deploymentFactory = Class.forName(className);
DeploymentFactory deploymentFactoryInstance =
   (DeploymentFactory) deploymentFactory.newInstance();

// Register the DeploymentFactory instance with the DeploymentFactoryManager.
dfm.registerDeploymentFactory(deploymentFactoryInstance);

// Provide WebSphere Application Server URL, user ID, and password.
// For more information, see the step that follows.
wsDM = dfm.getDeploymentManager(
   "deployer:WebSphere:myserver:8880", null, null);
```

   b. Write code that accesses the DeploymentManager instance for the product The product URL for deployment has the format

      `"deployer:WebSphere:host:port"`

      The example in the previous step, "deployer:WebSphere:myserver:8880", tries to connect to host *myserver* at port *8880* using the SOAP connector, which is the default.

      The URL for deployment can have an optional parameter *connectorType*. For example, to use the RMI connector to access *myserver*, code the URL as follows:

      `"deployer:WebSphere:myserver:2809?connectorType=RMI"`

2. Optional: Code a Java program that can customize or deploy Java EE applications or modules using the JSR-88 support provided by the product.
3. Start the deployed Java EE applications or stand-alone Java EE modules using the JSR-88 API used to start applications or modules.

### What to do next

Test the deployed applications or modules. For example, point a Web browser at the URL for a deployed application and examine the performance of the application. If necessary, update the application.

## Customizing modules using DConfigBeans

You can configure Java Platform, Enterprise Edition (Java EE) applications or standalone modules during deployment using the DConfigBean class in the Java EE Application Deployment API specification (JSR-88).

### Before you begin

This topic assumes that you are deploying (installing) Java EE modules on an application server provided by the product using the WebSphere Application Server support for JSR-88.

Read about the JSR-88 specification and using the DConfigBean class at http://java.sun.com/j2ee/tools/deployment/.

### About this task

The DConfigBean class in JSR-88 provides JavaBeans-based support for platform-specific configuration of Java EE applications and modules during deployment. Your code can inspect DConfigBean instances to get platform-specific configuration attributes. The DConfigBean instances provided by WebSphere Application Server contain a single attribute which has an array of java.util.Hashtable objects. The hashtable entries contain configuration attributes, for which your code can get and set values.

1. Write code that installs Java EE modules on an application server using JSR-88.
2. Write code that accesses DConfigBeans generated by the product during JSR-88 deployment. You (or a deployer) can then customize the accessed DConfigBeans instances. The following pseudocode shows how a Java EE tool provider can get DConfigBean instance attributes generated by the product during JSR-88 deployment and set values for the attributes.

```
import javax.enterprise.deploy.model.*;
import javax.enterprise.deploy.spi.*;
{
DeploymentConfiguration dConfig = ___; // Get from DeploymentManager
DDBeanRoot ddRoot = ___;               // Provided by J2EE tool

// Obtain root bean.
DConfigBeanRoot dcRoot = dConfig.getDConfigBeanRoot(dr);

// Configure DConfigBean.
configureDCBean (dcRoot);
}

// Get children from DConfigBeanRoot and configure each child.
method configureDCBean (DConfigBean dcBean)
{
   // Get DConfigBean attributes for a given archive.
   BeanInfo bInfo = Introspector.getBeanInfo(dcBean.getClass());
   IndexedPropertyDescriptor ipDesc =
      (IndexedPropertyDescriptor)bInfo.getPropertyDescriptors()[0];

   // Get the 0th table.
   int index = 0;
   Hashtable tbl = (Hashtable)
      ipDesc.getIndexedReadMethod().invoke
         (dcBean, new Object[]{new Integer(index)});

   while (tbl != null)
   {
      // Iterate over the hashtable and set values for attributes.

      // Set the table back into the DCBean.
      ipDesc.getIndexedWriteMethod().invoke
         (dcBean, new Object[]{new Integer(index), tbl});

      // Get the next entry in the indexed property
      tbl = (Hashtable)
         ipDesc.getIndexedReadMethod().invoke
            (dcBean, new Object[]{new Integer(++index)});
   }
}
```

## Enterprise application collection

Use this page to view and manage enterprise applications.

This page lists installed enterprise applications. System applications, which are central to the product, are not shown in the list because users cannot edit them. Examples of system applications include *isclite*, *managementEJB* and *filetransfer*.

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

To view the values specified for an application's configuration, click the application name in the list. The displayed application settings page shows the values specified. On the settings page, you can change existing configuration values and link to additional console pages that assist you in configuring the application.

To manage an installed enterprise application, enable the **Select** check box beside the application name in the list and click a button:

| Button | Resulting action |
|---|---|
| **Start** | Attempts to run the application. After the application starts up successfully, the state of the application changes to *Started* if the application starts up on all deployment targets, else the state changes to *Partial Start*. |
| **Stop** | Attempts to stop the processing of the application. After the application stops successfully, the state of the application changes to *Stopped* if the application stops on all deployment targets, else the state changes to *Partial Stop*. |
| **Install** | Opens a wizard that helps you deploy an application or a module such as a .jar, .war, .sar or .rar file onto a server or a cluster. |
| **Uninstall** | Deletes the application from the product configuration repository and deletes the application binaries from the file system of all nodes where the application modules are installed after the configuration is saved and synchronized with the nodes. |
| **Update** | Opens a wizard that helps you update application files deployed on a server. You can update the full application, a single module, a single file, or part of the application. If a new file or module has the same relative path as a file or module already existing on the server, the new file or module replaces the existing file or module. If the new file or module does not exist on the server, it is added to the deployed application. |
| **Remove File** | Deletes a file of the deployed application or module. **Remove File** deletes a file from the configuration repository and from the file system of all nodes where the file is installed. |
| **Export** | Accesses the Export Application EAR files page, which you use to export an enterprise application to an EAR file at a location of your choice. Use the **Export** action to back up a deployed application and to preserve its binding information. |
| **Export DDL** | Accesses the Export Application DDL files page, which you use to export DDL files (Table.ddl) in the EJB modules of an enterprise application to a location of your choice. |
| **Export File** | Accesses the Export a file from an application page, which you use to export a file of an enterprise application or module to a location of your choice.<br><br>If the browser does not prompt for a location to store the file, click **File** → **Save as** and specify a location to save the file that is shown in the browser. |

These buttons are not available when you access this page from an application server settings page. When this page is accessed from an application server settings page, it is entitled the Installed applications page.

When security is enabled, a separate application list is shown for each of your administrative roles. Supported roles include monitor, configurator, operator, administrator, deployer, and administrative security manager. For example, when you have the administrator role, the statement “You can administer the following resources” is shown followed by a list of applications that you can administer.

## Name

Specifies the name of the installed (or deployed) application. Application names must be unique within a cell and cannot contain an unsupported character.

## Application Status

Indicates whether the application deployed on the application server is started, stopped, or unknown.

| | | |
|---|---|---|
| | **Started** | Application is running. |

| | | |
|---|---|---|
| | **Partial Start** | Application is in the process of changing from a *Stopped* state to a *Started* state. Application is starting to run but is not fully running yet. Or, it cannot fully start because a server mapped to one or more application modules is stopped. |
| | **Stopped** | Application is not running. |
| | **Partial Stop** | Application is in the process of changing from a *Started* state to a *Stopped* state. Application has not stopped running yet. |
| | **Unknown** | Status cannot be determined. |
| | **Pending** | Status is temporarily unknown pending an event that a user did not initiate, such as pending an asynchronous call. |
| | **Not applicable** | Application does not provide information as to whether it is running. |

The status of an application on a Web server is always **Unknown**.

## Startup order

Specifies the order in which applications are started when the server starts. The application with the lowest startup order is started first.

This table column is available only when this page is accessed from an application server settings page; thus when this page is entitled the Installed applications page.

## Enterprise application settings

Use this page to configure an enterprise application.

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

If you have a JAX-WS Web service application installed, you also can click **Services** → **Service providers** → ***service_name*** or **Services** → **Service clients** → ***service_name***.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

### Name

Specifies a logical name for the application. An application name must be unique within a cell and cannot contain an unsupported character.

An application name cannot begin with a period (.), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

| **Unsupported characters** | | |
|---|---|---|
| **/** forward slash | **$** dollar sign | **'** single quote mark |
| **** backslash | **=** equal sign | **"** double quote mark |
| ***** asterisk | **%** percent sign | **\|** vertical bar |
| **,** comma | **+** plus sign | **<** left angle bracket |
| **:** colon | **@** at sign | **>** right angle bracket |
| **;** semi-colon | **#** hash mark | **&** ampersand (and sign) |
| **?** question mark | **]]>** No specific name exists for this character combination | |

| | |
|---|---|
| **Data type** | String |

## Application reference validation

Specifies whether the product examines the application references specified during application installation or updating and, if validation is enabled, warns you of incorrect references or fails the operation.

An application typically refers to resources using data sources for container managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the application is defined in the scope of the deployment target of that application.

The resource can be defined on the server, its node, cell or the cluster if the server belongs to a cluster. Select **Don't validate** for no resource validation, **Issue warnings** for warning messages about incorrect resource references, or **Stop installation if validation fails** to stop operations that fail as a result of incorrect resource references.

This **Application reference validation** setting is the same as the **Validate input off/warn/fail** field on the application installation and update wizards.

| | |
|---|---|
| **Data type** | String |
| **Default** | Issue warnings |

# Configuring enterprise application files

You can change the configuration of a Java Platform, Enterprise Edition (Java EE) application or module deployed on a server.

## Before you begin

You can change the contents of and deployment descriptors for an application or module before deployment, such as in an assembly tool. However, it is assumed that the module is already deployed on a server.

## About this task

Changing an application or module configuration consists of one or more of the following:
- Changing the settings of the application or module.
- Removing a file from an application or module.
- Updating the application or its modules.

This topic describes how to change the settings of an application or module using the administrative console.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

- View current settings of the application or module.

  Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** to access the enterprise application settings page.

Many application or module settings are available on other console pages that you can access by clicking links on the settings page for the enterprise application. For detailed information on the settings and allowed values, examine the online help for the console pages. When you installed the application or module, you specified most of the settings values.

- Map each module of your application to a target server.

  Specify the application servers or Web servers onto which to install modules of your application.
- Change how quickly your application starts compared to other applications or to the server.
- Configure the use of binary files.
- Change how your application or Web modules use class loaders.
- Map a virtual host for each Web module of your application. Configuring virtual hosts provides information on virtual hosts.
- Change application bindings or other settings of the application or module.
  1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → ***property_or_item_name***. From the application settings page, you can access console pages for further configuring of the application or module.
  2. Change the values for settings as needed, and click **OK**.
- Optional: Configure the application so it does not start automatically when the server starts. By default, an installed application starts when the server on which the application resides starts. You can configure the target mapping for the application so the application does not start automatically when the server starts. To start the application, you must then start it manually.
- If the installed application or module uses a resource adapter archive (RAR file), ensure that the **Classpath** setting for the RAR file enables the RAR file to find the classes and resources that it needs. Examine the **Classpath** setting on the console Resource adapter settings page.

### Results

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

### What to do next

Save changes to your administrative configuration.

## Application bindings

Before an application that is installed on an application server can start, all enterprise bean (EJB) references and resource references defined in the application must be bound to the actual artifacts (enterprise beans or resources) defined in the application server.

When defining bindings, you specify Java Naming and Directory Interface (JNDI) names for the referenceable and referenced artifacts in an application. The jndiName values specified for artifacts must be qualified lookup names. An example referenceable artifact is an EJB defined in an application. An example referenced artifact is an EJB or a resource reference used by the application.

Binding definitions are stored in the ibm-*xxx*-bnd.xml or ibm-*xxx*-bnd.xmi files of an application. Version 7.0 binding definitions support files with the suffix of XML for EJB 3.0 and Web 2.5 modules. Modules earlier than Java EE 5 continue to use binding definition files with the suffix of XMI as in previous versions of WebSphere Application Server. The *xxx* can be ejb-jar, web, application or application-client.

**Note:** For EJB 3.0 modules, you do not need to specify JNDI binding names for each of the home or business interfaces on your enterprise beans. If you do not explicitly assign bindings, the EJB container assigns default bindings. Further, binding definitions are stored in ibm-ejb-jar-bnd.xml. See EJB 3.0 application bindings overview.

This topic provides the following information about bindings:
- "Times when bindings can be defined"
- "Required bindings"
- "Other bindings that might be needed" on page 74

## Times when bindings can be defined

You can define bindings at the following times:

- During application development

  An application developer can create binding definitions in ibm-*xxx*-bnd.xml files for EJB 3.0 and Web 2.5 modules and in ibm-*xxx*-bnd.xmi files for pre-Java EE 5 modules. The application developer can create the files using a tool such as an IBM Rational developer tool or, for EJB 3.0 or Web 2.5 modules, using an XML editor or text editor. The developer then gives an enterprise application (.ear file) complete with bindings to an application assembler or deployer. When assembling the application, the assembler does not modify the bindings. Similarly, when installing the application onto a server supported by WebSphere Application Server, the deployer does not modify or override the bindings or generate default bindings unless changes to the bindings are necessary for successful deployment of the application.
- During application assembly

  An application assembler can define bindings in annotations or deployment descriptors of an application. Java EE 5 modules contain annotations in the source code. To declare an annotation, an application assembler precedes a keyword with an @ character ("at" sign). Bindings for pre-Java EE 5 modules are specified in the **WebSphere Bindings** section of a deployment descriptor editor. Modifying the deployment descriptors might change the binding definitions in the ibm-*xxx*-bnd.xmi files created when developing an application. After defining the bindings, the assembler gives the application to a deployer. When installing the application onto a server supported by WebSphere Application Server, the deployer does not modify or override the bindings or generate default bindings unless changes to the bindings are necessary to deploy the application.
- During application installation

  An application deployer or server administrator can modify the bindings when installing the application onto a server supported by WebSphere Application Server using the administrative console. New binding definitions can be specified on the installation wizard pages.

  Also, a deployer or administrator can select to generate default bindings during application installation. Selecting **Generate default bindings** during application installation instructs the product to fill in incomplete bindings in the application with default values. Existing bindings are not changed.

  **Note:** You cannot define or override bindings during application installation for application clients. You must define bindings for application client modules during assembly and store the bindings in the ibm-application-client-bnd.xmi file.
- During configuration of an installed application

  After an application is installed onto a server supported by WebSphere Application Server, an application deployer or server administrator can modify the bindings by changing values in administrative console pages such as those accessed from the settings page for the enterprise application.

## Required bindings

Before an application can be successfully deployed, bindings must be defined for references to the following artifacts:

**EJB JNDI names**

For each EJB 2.1 or earlier enterprise bean (EJB), you must specify a JNDI name. The name is used to bind an entry in the global JNDI name space for the EJB home object. An example JNDI

name for a *Product* EJB in a *Store* application might be `store/ejb/Product`. The binding definition is stored in the META-INF/ibm-ejb-jar-bnd.xmi file.

If a deployer chooses to generate default bindings when installing the application, the installation wizard assigns EJB JNDI names having the form *prefix*/*EJB_name* to incomplete bindings. The default prefix is ejb, but can be overridden. The *EJB_name* is as specified in the deployment descriptor `<ejb-name>` tag.

During and after application installation, EJB JNDI names can be specified on the Provide JNDI names for beans panel. After installation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **EJB JNDI names** in the administrative console.

You do not need to specify JNDI binding names for each of the EJB 3.0 home or business interfaces on your enterprise beans because the EJB container assigns default bindings. See EJB 3.0 application bindings overview.

**Data sources for entity beans**

Entity beans such as container-managed persistence (CMP) beans store persistent data in data stores. With CMP beans, an EJB container manages the persistent state of the beans. You specify which data store a bean uses by binding an EJB module or an individual enterprise bean to a data source. Binding an EJB module to a data source causes all entity beans in that module to use the same data source for persistence.

An example JNDI name for a *Store* data source in a *Store* application might be `store/jdbc/store`. For modules earlier than Java EE 5, the binding definition is stored in IBM binding files such as ibm-ejb-jar-bnd.xmi. A deployer can also specify whether authentication is handled at the container or application level.

WebSphere Application Server Version 7.0 supports CMP beans in EJB 2.x or 1.x modules. Version 7.0 does not support CMP beans in EJB 3.0 modules.

If a deployer chooses to generate default bindings when installing the application, the install wizard generates the following for incomplete bindings:

- For EJB 2.x .jar files, connection factory bindings based on the JNDI name and authorization information specified
- For EJB 1.1 .jar files, data source bindings based on the JNDI name, data source user name and password specified

The generated bindings provide default connection factory settings for each EJB 2.x .jar file and default data source settings for each EJB 1.1 .jar file in the application being installed. No bean-level connection factory bindings or data source bindings are generated unless they are specified in the custom strategy rule supplied during default binding generation.

During and after application installation, you can map data sources to 2.x entity beans on the 2.x CMP bean data sources panel and on the 2.x entity bean data sources panel. After installation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** in the administrative console, then select **2.x CMP bean data sources** or **2.x entity bean data sources**. You can map data sources to 1.x entity beans on the Map data sources for all 1.x CMP beans panel and on the Provide default data source mapping for modules containing 1.x entity beans panel. After installation, access console pages similar to those for 2.x CMP beans, except click links for 1.x CMP beans.

**Backend ID for EJB modules**

If an EJB .jar file that defines CMP beans contains mappings for multiple backend databases, specify the appropriate backend ID that determines which persister classes are loaded at run time.

Specify the backend ID during application installation. You cannot select a backend ID after the application is installed onto a server.

To enable backend IDs for individual EJB modules:

1. During application installation, select **Deploy enterprise beans** on the Select installation options panel. Selecting **Deploy enterprise beans** enables you to access the Provide options to perform the EJB Deploy panel.
2. On the Provide options to perform the EJB Deploy panel, set the database type to "" (null).

During application installation, if you select **Deploy enterprise beans** on the Select installation options panel and specify a database type for the EJB deployment tool on the Provide options to perform the EJB Deploy panel, previously defined backend IDs for all of the EJB modules are overwritten by the chosen database type.

The default database type is `DB2UDB_V81`.

The EJB deployment tool does not run during installation of EJB 3.0 modules.

**EJB references**

An enterprise bean (EJB) reference is a logical name used to locate the home interface of an enterprise bean. EJB references are specified during deployment. At run time, EJB references are bound to the physical location (global JNDI name) of the enterprise beans in the target operational environment. EJB references are made available in the java:comp/env/ejb Java naming subcontext.

The product assigns default JNDI values for or automatically resolves incomplete EJB 3.0 reference targets.

For each EJB 2.1 or earlier EJB reference, you must specify a JNDI name. An example JNDI name for a *Supplier* EJB reference in a *Store* application might be `store/ejb/Supplier`. The binding definition is stored in IBM binding files such as ibm-ejb-jar-bnd.xmi. When the referenced EJB is also deployed in the same application server, you can specify a server-scoped JNDI name. But if the referenced EJB is deployed on a different application server or if ejb-ref is defined in an application client module, then you should specify the global cell-scoped JNDI name.

If a deployer chooses to generate default bindings when installing the application, the install wizard binds EJB references as follows: If an `<ejb-link>` is found, it is honored. If the ejb-name of an EJB defined in the application matches the `ejb-ref` name, then that EJB is chosen. Otherwise, if a unique EJB is found with a matching home (or local home) interface as the referenced bean, the reference is resolved automatically.

During and after application installation, you can specify EJB reference JNDI names on the Map EJB references to beans panel. After installation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **EJB references** in the administrative console.

**Note:** To enable EJB reference targets to resolve automatically if the references are from EJB 2.1 or earlier modules or from Web 2.3 or earlier modules, select **Generate default bindings** on the Preparing for application installation panel or select **Allow EJB reference targets to resolve automatically** on the Select installation options, Map EJB references to beans, or EJB references console panels.

For more information, refer to EJB references .

**Resource references**

A resource reference is a logical name used to locate an external resource for an application. Resource references are specified during deployment. At run time, the references are bound to the physical location (global JNDI name) of the resource in the target operational environment. Resource references are made available as follows:

| Resource reference type | Subcontext declared in |
| --- | --- |
| Java DataBase Connectivity (JDBC) data source | java:comp/env/jdbc |
| JMS connection factory | java:comp/env/jms |
| JavaMail connection factory | java:comp/env/mail |

| Uniform Resource Locator (URL) connection factory | java:comp/env/url |
|---|---|

For each resource reference, you must specify a JNDI name. If a deployer chooses to generate default bindings when installing the application, the install wizard generates resource reference bindings derived from the `<res-ref-name>` tag, assuming that the java:comp/env name is the same as the resource global JNDI name.

During application installation, you can specify resource reference JNDI names on the Map resource references to references panel. Specify JNDI names for the resources that represent the logical names defined in resource references. You can optionally specify login configuration name and authentication properties for the resource. After specifying authentication properties, click **OK** to save the values and return to the mapping step. Each resource reference defined in an application must be bound to a resource defined in your WebSphere Application Server configuration. After installation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Resource references** in the administrative console to access the Resource references panel.

**Virtual host bindings for Web modules**

You must bind each Web module to a specific virtual host. The binding informs a Web server plug-in that all requests that match the virtual host must be handled by the Web application. An example virtual host to be bound to a *Store* Web application might be `store_host`. The binding definition is stored in IBM binding files such as WEB-INF/ibm-web-bnd.xmi.

If a deployer chooses to generate default bindings when installing the application, the install wizard sets the virtual host to `default_host` for each .war file.

During and after application installation, you can map a virtual host to a Web module defined in your application. On the Map virtual hosts for Web modules panel, specify a virtual host. The port number specified in the virtual host definition is used in the URL that is used to access artifacts such as servlets and JavaServer Pages (JSP) files in the Web module. For example, an external URL for a Web artifact such as a JSP file is http://*host_name*:*virtual_host_port*/*context_root*/*jsp_path*. After installation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Virtual hosts** in the administrative console.

**Message-driven beans**

For each message-driven bean, you must specify a queue or topic to which the bean will listen. A message-driven bean is invoked by a Java Messaging Service (JMS) listener when a message arrives on the input queue that the listener is monitoring. A deployer specifies a listener port or JNDI name of an activation specification as defined in a connector module (.rar file) under **WebSphere Bindings** on the Beans page of an assembly tool EJB deployment descriptor editor. An example JNDI name for a listener port to be used by a *Store* application might be `StoreMdbListener`. The binding definition is stored in IBM bindings files such as ibm-ejb-jar-bnd.xmi.

If a deployer chooses to generate default bindings when installing the application, the install wizard assigns JNDI names to incomplete bindings.

- For EJB 2.0 or later message-driven beans deployed as JCA 1.5-compliant resources, the install wizard assigns JNDI names corresponding to activationSpec instances in the form eis/MDB_ejb-name.
- For EJB 2.0 or later message-driven beans deployed against listener ports, the listener ports are derived from the message-driven bean `<ejb-name>` tag with the string `Port` appended.

During application installation using the administrative console, you can specify a listener port name or an activation specification JNDI name for every message-driven bean on the Bind listeners for message-driven beans panel. A listener port name must be provided when using the JMS providers: Version 5 default messaging, WebSphere MQ, or generic. An activation specification must be provided when the application's resources are configured using the default messaging provider or any generic J2C resource adapter that supports inbound messaging. If neither is specified, then a validation error is displayed after you click **Finish** on the Summary

panel. Also, if the module containing the message-driven bean is deployed on a 5.x deployment target and a listener port is not specified, then a validation error is displayed after you click **Next**.

After application installation, you can specify JNDI names and configure message-driven beans on console pages under **Resources** → **JMS** → **JMS providers** or under **Resources** → **Resource adapters**. For more information, refer to Choosing a messaging provider.

**Note:** You can only bind message driven-beans that are defined in an EJB 3.0 module to an activation specification. See EJB 3.0 application bindings overview.

**Message destination references**

A message destination reference is a logical name used to locate an enterprise bean in an EJB module that acts as a message destination. Message destination references exist only in J2EE 1.4 and later artifacts such as--

- J2EE 1.4 application clients
- EJB 2.1 projects
- 2.4 Web applications

If multiple message destination references are associated with a single message destination link, then a single JNDI name for an enterprise bean that maps to the message destination link, and in turn to all of the linked message destination references, is collected during deployment. At run time, the message destination references are bound to the administered message destinations in the target operational environment.

If a message destination reference and a message-driven bean are linked by the same message destination, both the reference and the bean should have the same destination JNDI name. When both have the same name, only the destination JNDI name for the message-driven bean is collected and applied to the corresponding message destination reference.

If a deployer chooses to generate default bindings when installing the application, the install wizard assigns JNDI names to incomplete message destination references as follows: If a message destination reference has a `<message-destination-link>`, then the JNDI name is set to ejs/message-destination-linkName. Otherwise, the JNDI name is set to eis/message-destination-refName.

### Other bindings that might be needed

Depending on the references in and artifacts used by your application, you might need to define bindings for references and artifacts not listed in this topic.

## Configuring application startup

You can configure the startup behavior of an application. The values set affect how quickly an application starts and what occurs when an application starts.

### Before you begin

This topic assumes that your application or module is already deployed on a server.

This topic also assumes that your application or module is configured to start automatically when the server starts. By default, an installed application starts when the server on which the application resides starts.

### About this task

This topic describes how to change the settings of an application or module using the administrative console.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Startup behavior** in the console navigation tree.

   The Startup behavior settings page is displayed.
2. Specify the startup order for the application.

   If your application starts automatically when its server starts, the value for **Startup order** on the Startup behavior settings page specifies the order in which applications are started when the server starts. The application with the lowest startup order, or starting weight, is started first. For example, specify `1` for **Startup order** for applications that you want started first, specify `2` for applications to be started next, and so.

   **Note:** For Session Initiation Protocol (SIP) applications, the `<load-on-startup>` tag in the sip.xml file affects the order in which applications are started. The value that you set for **Startup order** on the Startup behavior settings page determines the importance or weight of an application within a composition of SIP applications. For example, for the most important SIP application within a SIP application composition, specify `1` for **Startup order**. For the next most important SIP application within the composition, specify `2` for **Startup order**, and so on.
3. Specify whether the application must initialize fully before its server is considered started.

   If your application starts automatically when its server starts, **Launch application before server completes startup** specifies whether the application must initialize fully before its server is considered started. Background applications can be initialized on an independent thread, thus allowing the server startup to complete without waiting for the application. This setting applies only if the application is run on a Version 6.0 or later application server.
4. Specify whether to create MBeans for resources such as servlets or JavaServer Pages (JSP) files within an application when the application starts.

   The default for **Create MBeans for resources** is to create MBeans.

## Results

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

## What to do next

Save changes to your administrative configuration.

## Startup behavior settings

Use this page to configure when an application starts compared to other applications and to the server, and to configure whether MBeans for resources are created when an application starts.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Startup behavior**.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

***Startup order:***

Specifies the order in which applications are started when the server starts. The startup order is like a starting weight. The application with the lowest starting weight is started first.

**Note:** For Session Initiation Protocol (SIP) applications, the `<load-on-startup>` tag in the sip.xml file affects the order in which servlets within applications are started. The value that you set for **Startup order** on this Startup behavior console page determines the importance or weight of an application within a composition of SIP applications. For example, for the most important SIP application within a SIP application composition, specify `1` for **Startup order**. For the next most important SIP application within the composition, specify `2` for **Startup order**, and so on. For more information, see the JSR 116 specification.

| | |
|---|---|
| **Data type** | Integer |
| **Default** | 1 |
| **Range** | 0 to 2147483647 |

***Launch application before server completes startup:***

Specifies whether the application must initialize fully before the server starts.

The default setting of `false` indicates that server startup will not complete until the application starts.

A setting of `true` informs the product that the application might start on a background thread and thus server startup might continue without waiting for the application to start. Thus, the application might not be ready for use when the application server starts.

This setting applies only if the application is run on a Version 6.0 or later application server.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

***Create MBeans for resources:***

Specifies whether to create MBeans for various resources, such as servlets or JavaServer Pages (JSP) files, within an application when the application starts. The default is to create MBeans.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | true |

# Configuring binary location and use

You can designate where binary files (binaries) used by your application reside, whether the product distributes binaries for you automatically, and otherwise configure the use of binaries.

## Before you begin

This topic assumes that your application or module is already deployed on a server.

## About this task

This topic describes how to change the settings of an application or module using the administrative console.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node

where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Application binaries** in the console navigation tree. The Application binaries page is displayed.
2. Specify the directory to hold the application binaries.

   The default is ${APP_INSTALL_ROOT}/*cell_name*, where the ${APP_INSTALL_ROOT} variable is *profile_root*/installedApps. For example:

   `profile_root/installedApps/cell_name`

   Refer to "Application binary settings" for a detailed description of the **Location (full path)** setting.
3. Specify the bindings, extensions, and deployment descriptors that an application server uses.

   By default, an application server uses the bindings, extensions, and deployment descriptors located with the application deployment document, the deployment.xml file.

   To specify that the application server use the bindings, extensions, and deployment descriptors located in the application archive (EAR) file, select **Use configuration information in binary**. Select this setting for applications installed on 6.x or later deployment targets. This setting is not valid for applications installed on 5.x deployment targets.
4. Specify whether the product distributes application binaries automatically to other nodes on the cell.

   By default, **Enable binary distribution, expansion and cleanup post uninstallation** is selected and binaries are distributed automatically.

   If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

   **Note:** If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you.
5. Specify access permissions for binaries.
   a. Ensure that the **Enable binary distribution, expansion and cleanup post uninstallation** option is enabled. That option must be enabled to specify access permissions for binaries.
   b. For **File permissions**, specify a string that defines access permissions for binaries that are expanded in the named location.

      You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the multiple-selection list. List selections overwrite file permissions set in the text field.

      For details on **File permissions**, refer to "Application binary settings."
6. Click **OK**.

## Results

The application or module configuration is changed. The application or stand-alone Web module is restarted so the changes take effect.

## What to do next

Save changes to your administrative configuration.

## Application binary settings

Use this page to configure the location and distribution of application binary files.

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

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

***Location (full path):***

Specifies the directory to which the enterprise application archive (EAR) file is installed. This **Location** setting is the same as the **Directory to install application** field on the application installation and update wizards.

By default, an EAR file is installed in the *profile_root*/installedApps/*cell_name*/*application_name*.ear directory.

Setting options include the following:

- Do not specify a value and leave the field empty.

  The default value is ${APP_INSTALL_ROOT}/*cell_name*, where the ${APP_INSTALL_ROOT} variable is *profile_root*/installedApps. A directory having the EAR file name of the application being installed is appended to ${APP_INSTALL_ROOT}/*cell_name*. Thus, if you do not specify a directory, the EAR file is installed in the *profile_root*/installedApps/*cell_name*/*application_name*.ear directory.
- Specify a directory.

  If you specify a directory, the application is installed in *specified_path*/*application_name*.ear directory. A directory having the EAR file name of the application being installed is appended to the path that you specified for **Directory to install application** when installing the application. For example, if you installed Clock.ear and specify `C:/myapps` on Windows machines, the application is installed in the myapps/Clock.ear directory. The ${APP_INSTALL_ROOT} variable is set to the specified path.
- Specify `${APP_INSTALL_ROOT}/${CELL}` for the initial installation of the application.

  If you intend to export the application from one cell and later install the exported application on a different cell, specify the `${CELL}` variable for the initial installation of the application. For example, specify `${APP_INSTALL_ROOT}/${CELL}` for this setting. Exporting the application creates an enhanced EAR file that has the application and its deployment configuration. The deployment configuration retains the cell name of the initial installation in the destination directory unless you specify the `${CELL}` variable. Specifying the `${CELL}` variable ensures that the destination directory has the current cell name, and not the original cell name.

  **Note:** If an installation directory is not specified when an application is installed on a single-server configuration, the application is installed in ${APP_INSTALL_ROOT}/*cell_name*. When the server is made a part of a multiple-server configuration (using the addNode utility), the cell name of the new configuration becomes the cell name of the deployment manager node. If the `-includeapps` option is used for the addNode utility, then the applications that are installed prior to the addNode operation still use the installation directory ${APP_INSTALL_ROOT}/*cell_name*. However, an application that is installed after the server is added to the network configuration uses the default installation directory ${APP_INSTALL_ROOT}/*network_cell_name*. To move the application to the ${APP_INSTALL_ROOT}/*network_cell_name* location upon running the addNode operation, explicitly specify the installation directory as `${APP_INSTALL_ROOT}/${CELL}` during installation. In such a case, the application files can always be found under ${APP_INSTALL_ROOT}/*current_cell_name*.
- If the application has been exported and you want to install the exported EAR file in a different cell or location, specify ${APP_INSTALL_ROOT}/*cell_name*/*application_name*.ear if you did not specify `${APP_INSTALL_ROOT}/${CELL}` for the initial installation.

  The exported EAR file is an enhanced EAR file that has the application and its deployment configuration. The deployment configuration retains the value for **Directory to install application** that

was used for the previous installation of the application. Unless you specify a different value, the enhanced EAR file will be installed to the same directory as for the previous installation.

If you did not specify the `${CELL}` variable during the initial installation, the deployment configuration uses the cell name of the initial installation in the destination directory. If you are installing on a different cell, specify ${APP_INSTALL_ROOT}/*cell_name*/*application_name*.ear, where *cell_name* is the name of the cell to which you want to install the enhanced EAR file. If you do not designate the current cell name, *cell_name* will be the original cell name even though you are installing the enhanced EAR file on a cell that has a different name.

- Specify an absolute path or a use pathmap variable.

  You can specify an absolute path or use a pathmap variable such as `${MY_APPS}`. You can use a pathmap variable in any installation.

| | |
|---|---|
| **Data type** | String |
| **Units** | Full path name |

***Use configuration information in binary:***

Specifies whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the deployment.xml file (default), or those located in the EAR file.

The default (false) is to use the binding, extensions, and deployment descriptors located in deployment.xml. To use the binding, extensions, and deployment descriptors located in the EAR file, enable this setting (true).

This **Use configuration information in binary** setting is the same as the **Use binary configuration** field on the application installation and update wizards. Select this setting for applications installed on 6.x or later deployment targets only. This setting is not valid for applications installed on 5.x deployment targets.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

***Enable binary distribution, expansion and cleanup post uninstallation:***

Specifies whether the product expands application binaries in the installation location during installation and deletes application binaries during uninstallation. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified.

On single-server installations, the binaries are deleted when you uninstall and save changes to the configuration.

If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

**Note:** If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you.

This **Enable binary distribution, expansion and cleanup post uninstallation** setting is the same as the **Distribute application** field on the application installation and update wizards.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | true |

***File permissions:***

Specifies access permissions for application binaries for installed applications that are expanded to the directory specified.

The **Enable binary distribution, expansion and cleanup post uninstallation** option must be enabled to specify file permissions.

You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the multiple-selection list. List selections overwrite file permissions set in the text field.

You can set one or more of the following file permission strings in the list. Selecting multiple options combines the file permission strings.

| Multiple-selection list option | File permission string set |
| --- | --- |
| **Allow all files to be read but not written to** | `.*=755` |
| **Allow executables to execute** | `.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755` |
| **Allow HTML and image files to be read by everyone** | `.*\.htm=755#.*\.html=755#.*\.gif=755#.*\.jpg=755` |

Instead of using the multiple-selection list to specify file permissions, you can specify a file permission string in the text field. File permissions use a string that has the following format:

*file_name_pattern=permission#file_name_pattern=permission*

where *file_name_pattern* is a regular expression file name filter (for example, `.*\\.jsp` for all JSP files), *permission* provides the file access control lists (ACLs), and *#* is the separator between multiple entries of *file_name_pattern* and *permission*. If `#` is a character in a *file_name_pattern* string, use `\#` instead.

If multiple file name patterns and file permissions in the string match a uniform resource identifier (URI) within the application, then the product uses the most stringent applicable file permission for the file. For example, if the file permission string is `.*\\.jsp=775#a.*\\.jsp=754`, then the abc.jsp file has file permission 754.

**Note:** Using regular expressions for file matching pattern compares an entire string URI against the specified file permission pattern. You must provide more precise matching patterns using regular expressions as defined by Java programming API. For example, suppose the following directory and file URIs are processed during a file permission operation:

| | |
| --- | --- |
| 1 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war` |
| 2 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp` |
| 3 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF/MANIFEST.MF` |
| 4 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/WEB-INF/classes/MyClass.class` |
| 5 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/mydir/MyClass2.class` |
| 6 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF` |

The file pattern matching results are:
- `MyWarModule.war` does not match any of the URIs
- `.*MyWarModule.war.*` matches all URIs

- `.*MyWarModule.war$` matches only URI 1
- `.*\\.jsp=755` matches only URI 2
- `.*META-INF.*` matches URIs 3 and 6
- `.*MyWarModule.war/.*/.*\.class` matches URIs 4 and 5

If you specify a directory name pattern for **File permissions**, then the directory permission is set based on the value specified. Otherwise, the **File permissions** value set on the directory is the same as its parent. For example, suppose you have the following file and directory structure:

```
/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
```

and you specify the following file pattern string:

```
.*MyApp.ear$=755#.*\.jsp=644
```

The file pattern matching results are:
- Directory MyApp.ear is set to 755
- Directory MyWarModule.war is set to 755
- Directory MyWarModule.war is set to 755

**Note:** Regardless of the operation system, always use a forward slash (/) as a file path separator in file patterns.

Access permissions specified here are at the application level. You can also specify access permissions for application binaries in the node level configuration. The node level file permissions specify the maximum (most lenient) permissions that can be given to application binaries. Access permissions specified here at application level can only be the same as or more restrictive than those specified at the node level.

This setting is the same as the **File permission** field on the application installation and update wizards.

| | |
|---|---|
| **Data type** | String |

***Application build level:***

Specifies an uneditable string that identifies the build version of the application.

| | |
|---|---|
| **Data type** | String |

## Configuring the use of class loaders by an application

You can configure whether your application and Web modules use their own class loaders to load classes or use different class loaders, as well as configure the reloading of classes when application files are updated. Class loaders enable an application to access repositories of available classes and resources.

### Before you begin

This topic assumes that your application or module is already deployed on a server.

### About this task

Class loaders affect whether your application and its modules find the resources that they need to run effectively. You can select whether your application and Web modules use their own class loaders to load classes, or use a parent class loader.

An application class loader groups enterprise bean (EJB) modules, shared libraries, resource adapter archives (RAR files), and dependency Java archive (JAR) files associated to an application. Dependency JAR files are JAR files that contain code which can be used by both enterprise beans and servlets.

An application class loader is the parent of a Web application archive (WAR) class loader. By default, a Web module has its own WAR class loader to load the contents of the Web module. The WAR class-loader policy value of an application class loader determines whether the WAR class loader or the application class loader is used to load the contents of the Web module.

You can also select whether classes are reloaded when application files are updated. For enterprise bean (EJB) modules or any non-Web modules, enabling class reloading causes the application server run time to stop and start the application to reload application classes. For Web modules such as servlets and JavaServer Pages (JSP) files, a Web container reloads a Web module only when the IBM extension reloadingEnabled in the ibm-web-ext.xmi file is set to `true`.

To configure use of class loaders by your application and Web modules, use the Class loading and update detection page of the administrative console.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Class loading and update detection** to access the settings page for an application class loader.
2. Specify whether to reload application classes when the application or its files are updated.

   By default, class reloading is not enabled. Select **Override class reloading settings for Web and EJB modules** to choose to reload application classes. You might specify different values for EJB modules and for Web modules such as servlets and JavaServer Pages (JSP) files.
3. Specify the number of seconds to scan the application's file system for updated files.

   The value specified for **Polling interval for updated files** takes effect only if class reloading is enabled. The default is the value of the reloading interval attribute in the IBM extension (META-INF/ibm-application-ext.xmi) file of the enterprise application (EAR file). You might specify different values for EJB modules and for Web modules such as servlets and JSP files.

   To enable reloading, specify an integer value that is greater than zero (for example, 1 to 2147483647).

   To disable reloading, specify zero (0).
4. Specify the class loader order for the application.

   The application class loader order specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The default is to search in the parent class loader before searching in the application class loader to load a class.

   Select either of the following values for **Class loader order**:

| Option | Description |
| --- | --- |
| **Classes loaded with parent class loader first** | Causes the class loader to search in the parent class loader first to load a class. This value is the standard for Development Kit class loaders and WebSphere Application Server class loaders. |

| Option | Description |
| --- | --- |
| **Classes loaded with local class loader first (parent last)** | Causes the class loader to search in the application class loader first to load a class. By specifying `Classes loaded with local class loader first (parent last)`, your application can override classes contained in the parent class loader.<br>**Note:** Specifying the `Classes loaded with local class loader first (parent last)` value might result in LinkageErrors or ClassCastException messages if you have mixed use of overridden classes and non-overridden classes. |

5. Specify whether to use a single or multiple class loaders to load Web application archives (WAR files) of your application.

   By default, Web modules have their own WAR class loader to load the contents of the WEB-INF/classes and WEB-INF/lib directories. The default WAR class loader value is `Class loader for each WAR file in application`, which uses a separate class loader to load each WAR file. Setting the value to `Single class loader for application` causes the application class loader to load the Web module contents as well as the EJB modules, shared libraries, RAR files, and dependency JAR files associated to the application. The application class loader is the parent of the WAR class loader.

   Select either of the following values for **WAR class loader policy**:

| Option | Description |
| --- | --- |
| **Class loader for each WAR file in application** | Uses a different class loader for each WAR file. |
| **Single class loader for application** | Uses a single class loader to load all of the WAR files in your application. |

6. Click **OK**.

## Results

The application or module configuration is changed. The application or standalone Web module is restarted so the changes take effect.

## What to do next

Save changes to your administrative configuration.

## Class loading and update detection settings

Use this page to configure use of class loaders by an application.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Class loading and update detection**.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

***Override class reloading settings for Web and EJB modules:***

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

Select **Override class reloading settings for Web and EJB modules** to set reloadEnabled to `true` in the deployment.xml file for the application. If an application's class definition changes, the application server run time stops and starts the application to reload application classes.

Reloading settings in the deployment.xml file override the reloading settings for all Web and EJB modules that can be defined in ibm-web-ext.xmi and META-INF/ibm-application-ext.xmi files.

For JavaServer Pages (JSP) files in a Web module, a Web container reloads JSP files only when the IBM extension jspReloadingEnabled in the jspAttributes of the ibm-web-ext.xmi file is set to `true`. You can enable JSP reloading during deployment on the JSP Reload Options panel.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

***Polling interval for updated files:***

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

This **Polling interval for updated files** setting is the same as the **Reload interval in seconds** field on the application installation and update wizards.

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

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

| | |
|---|---|
| **Data type** | Integer |
| **Units** | Seconds |
| **Default** | 3 |

***Class loader order:***

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for development kit class loaders and WebSphere Application Server class loaders is `Classes loaded with parent class loader first`. By specifying `Classes loaded with local class loader first (parent last)`, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.

The options are `Classes loaded with parent class loader first` and `Classes loaded with local class loader first (parent last)`. The default is to search in the parent class loader before searching in the application class loader to load a class.

For your application to use the default configuration of Jakarta Commons Logging in WebSphere Application Server, set this application class loader mode to `Classes loaded with parent class loader first`. For your application to override the default configuration of Jakarta Commons Logging in WebSphere Application Server, your application must provide the configuration in a form supported by Jakarta Commons Logging and this class loader mode must be set to `Classes loaded with local class loader first (parent last)`. Also, to override the default configuration, set the class loader mode for each Web module in your application so that the correct logger factory loads.

| | |
|---|---|
| **Data type** | String |
| **Default** | Classes loaded with parent class loader first |

***WAR class loader policy:***

Specifies whether to use a single class loader to load all WAR files of the application or to use a different class loader for each WAR file.

The options are `Class loader for each WAR file in application` and `Single class loader for application`. The default is to use a separate class loader to load each WAR file.

| | |
|---|---|
| **Data type** | String |
| **Default** | Class loader for each WAR file in application |

## Manage modules settings

Use this panel to specify deployment targets where you want to install the modules that are contained in your application. Modules can be installed on the same deployment target or dispersed among several deployment targets.

On single-server products, a deployment target can be an application server or Web server.

To view this administrative console panel, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Manage modules**. This panel is the similar to the Map modules to servers panel on the application installation and update wizards.

On this panel, each **Module** must map to one or more targets, identified under **Server**. To change a mapping:

1. In the list of mappings, select each module that you want mapped to the same target or targets.
2. From the **Clusters and servers** list, select one or more targets. Select only appropriate deployment targets for a module. You cannot install modules that use WebSphere Application Server Version 7.x features on a Version 6.x or 5.x target server.

   Use the Ctrl key to select multiple targets. For example, to have a Web server serve your application, press the Ctrl key and then select an application server and the Web server together. The product generates the plug-in configuration file, plugin-cfg.xml, for that Web server based on the applications which are routed through it.
3. Click **Apply**.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

If you accessed this Manage modules panel from a console enterprise application page for an already installed application, you can also use this panel to view and manage modules in your application.

To view the values specified for a module configuration, click the module name in the list. The displayed module settings page shows the values specified. On the settings page, you can change existing configuration values and link to additional console pages that assist you in configuring the module.

To manage a module, select the module name in the list and click a button:

| Button | Resulting action |
| --- | --- |
| **Remove** | Removes the selected module from the deployed application. The module is deleted from the application in the configuration repository and also from all of the nodes where the application is installed and running or expected to run. |
| **Update** | Opens a wizard that helps you update modules in an application. If a module has the same URI as a module already existing in the application, the new module replaces the existing module. If the new module does not exist in the application, it is added to the deployed application. |
| **Remove File** | Deletes a file from a module of a deployed application. |
| **Export File** | Accesses the Export a file from an application page, which you use to export a file of an enterprise application or module to a location of your choice.<br><br>If the browser does not prompt for a location to store the file, click **File** → **Save as** and specify a location to save the file that is shown in the browser. |

## Clusters and servers

Lists the names of available deployment targets. This list is the same for every application that is installed in the cell.

From this list, select only appropriate deployment targets for a module. You must install an application, enterprise bean (EJB) module, Session Initiation Protocol (SIP) module (SAR), or Web module on a Version 7.x target under any of the following conditions:
- The module supports Java Platform, Enterprise Edition (Java EE) 5.
- The module calls a 7.x runtime application programming interface (API).
- The module uses a 7.x product feature.

If a module supports J2EE 1.4, then you must install the module on a Version 6.x or 7.x deployment target. Modules that call a 6.1.x API or use a 6.1.x feature can be installed on a 6.1.x or 7.x deployment target. Modules that call a 6.0.x API or use a 6.0.x feature can be installed on a 6.0.x, 6.1.x or 7.x deployment target. Modules that require 6.1.x feature pack functionality can be installed on a 6.1.x deployment target that has been enabled with that feature pack or on a 7.x deployment target.

You can install an application or module developed for a Version 5.x product on a 5.x, 6.x or 7.x deployment target.

## Module

Specifies the name of a module in the installed (or deployed) application.

## URI

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

## Module type

Specifies the type of module, for example, a Web module or EJB module.

This setting is shown on the Manage modules panel accessed from a console enterprise application page.

## Server

Specifies the name of each deployment target to which the module currently is mapped.

To change the deployment targets for a module, select one or more targets from the **Clusters and servers** list and click **Apply**. The new mapping replaces the previous mapping.

## Mapping modules to servers

Each module of a deployed application must be mapped to one or more target servers. The target server can be an application server or Web server.

### Before you begin

You can map modules of an application or stand-alone Web module to one or more target servers during or after application installation using the console. This topic assumes that the module is already installed on a server and that you want to change the mappings.

Before you change a mapping, check the deployment targets. You must specify an appropriate deployment target for a module. Modules that use Version 7.x features cannot be installed onto Version 6.x or 5.x target servers.

### About this task

During application installation, different deployment targets might have been specified.

You use the Manage modules panel of the administrative console to view and change mappings. This panel is displayed during application installation using the console and, after the application is installed, can be accessed from the enterprise application settings page.

On the Manage modules panel, specify target servers where you want to install the modules contained in your application. Modules can be installed on the same application server or dispersed among several application servers. Also, specify the Web servers as targets that will serve as routers for requests to your application. The plug-in configuration file, plugin-cfg.xml, for each Web server is generated based on the applications which are routed through it.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Manage modules** in the console navigation tree.

   The Manage modules panel is displayed.
2. Examine the list of mappings.

   Ensure that each **Module** entry is mapped to one or more targets, identified under **Server**.
3. Change a mapping as needed.
   a. Select each module that you want mapped to the same targets.

      In the list of mappings, select check boxes for the modules.
   b. From the **Clusters and servers** list, select one or more targets.

      Select only appropriate deployment targets for a module. You cannot install modules that use WebSphere Application Server Version 7.x features on a Version 6.x or 5.x target server.

      Use the Ctrl key to select multiple targets. For example, to have a Web server serve your application, use the Ctrl key to select an application server and the Web server together to have the plugin-cfg.xml plug-in configuration file for that Web server generated based on the applications that are routed through it.
   c. Click **Apply**.
4. Repeat steps 2 and 3 until each module maps to the desired targets.
5. Click **OK**.

## Results

The application or module configurations are changed. The application or stand-alone Web module is restarted so the changes take effect.

## Example

To install an application that has modules which support Java Platform, Enterprise Edition (Java EE) 5 to two servers, do the following:

1. Click the **Select All** icon to select all of the modules in the application.
2. While pressing Ctrl, select two Version 7 application servers from the **Clusters and servers** list.
3. Click **Apply**.
4. Click **OK**.

## What to do next

Save changes to your administrative configuration.

# Mapping virtual hosts for Web modules

A virtual host must be mapped to each Web module of a deployed application. Web modules can be installed on the same virtual host or dispersed among several virtual hosts.

## Before you begin

You can map a virtual host to a Web module during or after application installation using the console. This article assumes that the Web module is already installed on a server and that you want to change the mappings.

Before you change a mapping, check the virtual hosts definitions. You can install a Web module on any defined virtual host. To view information on previously defined virtual hosts, click **Environment** → **Virtual hosts** in the administrative console. Virtual hosts enable you to associate a unique port with a module or application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host alias is used in the URL that is used to access artifacts such as servlets and JavaServer Pages (JSP) files in a Web module. For example, the alias `myhost:8080` is the `host_name:port_number` portion of the URL `http://myhost:8080/servlet/snoop`.

## About this task

During application installation, a virtual host other than the one you want mapped to your Web module might have been specified.

The default virtual host setting usually is `default_host`, which provides several port numbers through its aliases:

**80** An internal, insecure port used when no port number is specified

**9080** An internal port

**9443** An external, secure port

Unless you want to isolate your Web module from other modules or resources on the same node (physical machine), `default_host` is a suitable virtual host for your Web module.

In addition to `default_host`, the product provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the Web module relates to system administration.

Use the Virtual hosts page of the administrative console to view and change mappings. This page is displayed during application installation using the console and, after the application is installed, can be accessed from the settings page for an enterprise application.

On the Virtual hosts page, specify a virtual host for each Web module. Web modules of an application can be installed on the same virtual host or on different virtual hosts.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Virtual hosts** in the console navigation tree. The Virtual hosts page is displayed.
2. Examine the list of mappings. Ensure that each **Web module** entry has the desired virtual host mapped to it, identified under **Virtual host**.
3. Change the mappings as needed.
   a. Select each Web module that you want mapped to a particular virtual host. In the list of mappings, place a check mark in the **Select** check boxes beside the Web modules.
   b. From the **Virtual host** drop-down list, select the desired virtual host. If you selected more than one virtual host in step 1:
      1) Expand **Apply Multiple Mappings**.
      2) Select the desired virtual host from the **Virtual host** drop-down list.
      3) Click **Apply**.
4. Repeat steps 2 and 3 until a desired virtual host is mapped to each Web module.
5. Click **OK**.

## Results

The application or Web module configurations are changed. The application or standalone Web module is restarted so the changes take effect.

## What to do next

After mapping virtual hosts, do the following:

1. Regenerate the plug-in configuration file.
   a. Click **Servers** → **Server Types** → **Web servers**.
   b. Select the Web server for which you want to generate a plug-in.
   c. Click **Generate Plug-in**.
2. Save changes to your administrative configuration.

## Virtual hosts settings

Use this panel to specify virtual hosts for Web modules contained in your application. Web modules can be installed on the same virtual host or dispersed among several virtual hosts.

To view this administrative console panel, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Virtual hosts**. This panel is the same as the **Map virtual hosts for Web modules** panel on the application installation and update wizards.

On this panel, each Web module must map to a previously defined virtual host, identified under **Virtual host**. You can see information on previously defined virtual hosts by clicking **Environment** → **Virtual hosts** in the administrative console. Virtual hosts enable you to associate a unique port with a module or application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host alias is used in the URL that is used to access artifacts such as servlets and JavaServer Pages (JSP) files in a Web module. For example, the alias `myhost:8080` is the `host_name:port_number` portion of the URL http://myhost:8080/servlet/snoop.

The default virtual host setting usually is `default_host`, which provides several port numbers through its aliases:

**80** An internal, insecure port used when no port number is specified

**9080** An internal port
**9443** An external, secure port

Unless you want to isolate your Web module from other modules or resources on the same node (physical machine), `default_host` is a suitable virtual host for your Web module.

In addition to `default_host`, the product provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the Web module relates to system administration.

To change a mapping:

1. In the list of mappings, select the **Select** check box beside each Web module that you want mapped to a particular virtual host.
2. From the **Virtual host** drop-down list, select the desired virtual host. If you selected more than one virtual host in step 1:
   a. Expand **Apply Multiple Mappings**.
   b. Select the desired virtual host from the **Virtual Host** drop-down list.
   c. Click **Apply**.
3. Click **OK**.

**Note:** If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

***Web module:***

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

***Virtual host:***

Specifies the name of the virtual host to which the Web module is currently mapped.

Expanding the drop-down list displays a list of previously defined virtual hosts. To change a mapping, select a different virtual host from the list.

Do not specify the same virtual host for different Web modules that have the same context root and are deployed on targets belonging to the same node even if the Web modules are contained in different applications. Specifying the same virtual host causes a validation error.

## Mapping properties for a custom login or trusted connection configuration

Use this page to view and manage the mapping properties for a custom login configuration or a trusted connection configuration.

To access the administrative console panel, complete the following steps:

1. Click **Applications > Application types> WebSphere enterprise applications >** *application_name*.
2. From Enterprise JavaBeans™ Properties, click **Map data sources for all 2.x CMP beans**.
3. For container authorization, modify the authorization type by selecting your Enterprise JavaBeans(EJB) module and selecting **Container** from the Resource authorization menu.
4. Click **Apply**.

5. From Specify authentication method, select **Use custom login configuration** or **Use trusted connections** and the name of the application login configuration.
6. Select the name of your EJB module.
7. Click **Apply**.
8. Click **Mapping properties** in the Resource authorization column. This property is not available until after you click **Apply** in the previous step.

### Name

Specifies the name for the mapping property.

Do not use the `MAPPING_ALIAS` property name because the name is reserved by the product.

### Value

Specifies the value paired with the specified name.

### Description

Specifies additional information about the name and value pair.

## Viewing deployment descriptors

A deployment descriptor is an extensible markup language (XML) file that specifies configuration and container options for an application or module.

### Before you begin

This topic assumes that you have installed an application or module on a server and that you want to view its deployment descriptor.

### About this task

When you create a Java 2 Platform, Enterprise Edition (J2EE) application or module in an assembly tool, the assembly tool creates deployment descriptor files for the application or module. Java Platform, Enterprise Edition (Java EE) 5 applications or modules might use annotations instead of deployment descriptors.

After an application or module is installed on a server, you can view its deployment descriptor in the administrative console. You cannot view Java EE 5 annotations.

Unless an application supports Java EE 5, an enterprise archive (EAR) file must contain an application.xml file. The application.xml identifies each module of an application. A Java EE 5 application is not required to provide an application.xml file in the EAR file. When an application.xml file does not exist, the product examines the Java archive (JAR) file contents to determine whether the JAR file is an enterprise bean (EJB) module or an application client module. A JAR file should not contain more than one deployment descriptor in it. When an ejb-jar.xml file is found in a JAR file, the product considers it an EJB module. If an ejb-jar.xml file is not found and an application-client.xml is found, the product considers the JAR file to be an application client module. If both ejb-jar.xml and application-client.xml files exist in the JAR file, the product might consider a JAR file intended to be an application client module to be an EJB module or a JAR file intended to be an EJB module to be an application client module. A JAR file should not contain more than one kind of deployment descriptor.

1. Access a deployment descriptor view.

   Click the navigational option stated in **Accessing a console view** to view the deployment descriptor for a given module:

| Module | Deployment descriptor file | Accessing a console view |
| --- | --- | --- |
| Enterprise application | application.xml | **Applications → Application Types → WebSphere enterprise applications → *application_name* → View deployment descriptor** |
| Web application | WEB-INF/web.xml | **Applications → Application Types → WebSphere enterprise applications → *application_name* → Manage modules → *module_name* → View deployment descriptor** |
| | WEB-INF/portlet.xml | **Applications → Application Types → WebSphere enterprise applications → *application_name* → Manage modules → *module_name* → View portlet deployment descriptor** |
| Enterprise bean | ejb-jar.xml | **Applications → Application Types → WebSphere enterprise applications → *application_name* → Manage modules → *module_name* → View deployment descriptor** |
| Application client | application-client.xml | No console view |
| Web service | webservices.xml | **Applications → Application Types → WebSphere enterprise applications → *application_name* → Manage modules → *module_name* >**<br>• **View Web services client deployment descriptor extension**<br>• **View Web services server deployment descriptor**<br>• **View Web services server deployment descriptor extension**<br><br>Viewing Web services deployment descriptors in the administrative console describes the views. |
| Resource adapter | ra.xml | **Resources → Resource Adapters → Resource adapters → *module_name* → View deployment descriptor** |

2. Click **Expand All** to view the deployment descriptor contents.

## Results

The deployment descriptor for the application or module is displayed.

## Example

The deployment descriptor for the product DefaultApplication follows:

```
<application id="Application_ID" >
  <display-name> DefaultApplication.ear</display-name>
  <description> This is the IBM WebSphere Application Server Default Application.</<description>
  <module id="WebModule_1" >
    <web>
      <web-uri> DefaultWebApplication.war</web-uri>
      <context-root> /</context-root>
    </web>
  </module>
  <module id="EjbModule_1" >
    <ejb> Increment.jar</ejb>
  </module>
  <security-role id="SecurityRole_1204342979281" >
    <description> All Authenticated users role.</description>
    <role-name> All Role</role-name>
  </security-role>
</application>
```

## What to do next

After displaying a deployment descriptor on the console page, do the following:

1. Examine the deployment descriptor contents, including any configurations that it has for bindings, security roles, references to other resources, or Java Naming and Directory Interface (JNDI) names.

   For example, examine the JAR files of your Java EE 5 module to ensure that each JAR file does not contain more than one kind of deployment descriptor. If a JAR file contains more than one kind of deployment descriptor, proceed to the next step and remove the extraneous deployment descriptor. Thus, if both ejb-jar.xml and application-client.xml files exist in a JAR file, remove the deployment descriptor that your module does not need.
2. Change a deployment descriptor as needed.

   You can edit a deployment descriptor file manually. However, it is preferable to edit a deployment descriptor using the console or in an assembly tool deployment descriptor editor to ensure that the deployment descriptor has valid properties and that its references contain appropriate values.

If your EJB 3.0 or Web 2.5 module does not have a `metadata-complete` attribute or the `metadata-complete` attribute is set to `false`, you can instruct the product to write the entire module deployment descriptor, including deployment information from annotations, to XML format. On the Metadata for modules page, select **metadata-complete attribute**.

**Note:** If your Java EE 5 application uses annotations and a shared library, do not select **metadata-complete attribute**. When your application uses annotations and a shared library, setting the `metadata-complete` attribute to `true` causes the product to incorrectly represent an `@EJB` annotation in the deployment descriptor as `<ejb-ref>` rather than `<ejb-local-ref>`. For Web modules, setting the `metadata-complete` attribute to `true` might cause InjectionException errors. If you must set the `metadata-complete` attribute to `true`, avoid errors by not using a shared library, by placing the shared library in either the classes or lib directory of the application server, or by fully specifying the metadata in the deployment descriptors.

## Metadata for module settings

Use this page to instruct a Java Platform, Enterprise Edition (Java EE) 5 enterprise bean (EJB) or Web module deployment descriptor to ignore annotations that specify deployment information.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Metadata for modules**. This page is the same as the Metadata for modules page on the application installation and update wizards.

**Note:** If your application contains EJB 3.0 or Web 2.5 modules, you can select to lock the deployment descriptor of one or more of the EJB 3.0 or Web 2.5 modules on the Metadata for modules page. If you set the `metadata-complete` attribute to `true` and lock deployment descriptors, the product writes the complete module deployment descriptor, including deployment information from annotations, to XML format.

Annotations are a standard mechanism of adding metadata to Java classes. You can use metadata to simplify development and deployment of Java EE 5 artifacts. Prior to the introduction of Java language annotations, deployment descriptors were the standard mechanism used by Java EE components. These deployment descriptors were mapped to XML format, which facilitated their persistence. If you select to lock deployment descriptors, the product merges Java EE 5 annotation-based metadata with the XML-based existing deployment descriptor metadata and persists the result.

### Module

Specifies the name of a module in the installed (or deployed) application.

| | |
|---|---|
| **Data type** | String |

### URI

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

| | |
|---|---|
| **Data type** | String |

### metadata-complete attribute

Specifies whether to write the complete module deployment descriptor, including deployment information from annotations, to extensible markup language (XML) format.

The default is not to write out a module deployment descriptor.

If your EJB 3.0 or Web 2.5 module does not have a `metadata-complete` attribute or the `metadata-complete` attribute is set to `false`, you can select a check box and instruct the product to write out a module deployment descriptor.

**Note:** If your Java EE 5 application uses annotations and a shared library, do not select **metadata-complete attribute**. When your application uses annotations and a shared library, setting the `metadata-complete` attribute to `true` causes the product to incorrectly represent an `@EJB` annotation in the deployment descriptor as `<ejb-ref>` rather than `<ejb-local-ref>`. For Web modules, setting the `metadata-complete` attribute to `true` might cause InjectionException errors. If you must set the `metadata-complete` attribute to `true`, avoid errors by not using a shared library, by placing the shared library in either the classes or lib directory of the application server, or by fully specifying the metadata in the deployment descriptors.

After you select a check box, you cannot deselect (clear) the check box and the module is no longer shown in the list of modules on this page. If you select all of the check boxes, the link to this page is no longer shown on the enterprise application settings page.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false (deselected) |

## Starting or stopping enterprise 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*).

### Before you begin

This topic assumes that the Java Platform, Enterprise Edition (Java EE) application is installed on a server. By default, the application starts automatically when the server starts.

### About this task

You can start and stop applications manually using the following:
- Administrative console
- startApplication and stopApplication attributes of the AdminControl object with the wsadmin tool
- startApplication and stopApplication administrative jobs of the AdminTask.submitJob -jobType object with the wsadmin tool
- Java programs that use ApplicationManager or AppManagement MBeans

This topic describes how to use the administrative console to start or stop an application.

**Note:** This topic applies to applications that do not contain Java Application Programming Interface (API) for XML-Based Web Services (JAX-WS) service providers. To stop or start applications that contain JAX-WS service providers, use the Service providers page accessed by clicking **Services** → **Service providers**. To start a service provider application, select a service and click **Start**

**Application**. To stop a service provider application, select a service and click **Stop Application**. Then, on the Stop application page, click **OK** to stop all modules in the application, including other services such as enterprise beans and servlets.

1. Go to the Enterprise applications page. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree.
2. Select the check box for the application you want started or stopped.
3. Click a button:

| Option | Description |
| --- | --- |
| **Start** | Runs the application and changes the state of the application to *Started*. The status is changed to *partially started* if not all servers on which the application is deployed are running. |
| **Stop** | Stops the processing of the application and changes the state of the application to *Stopped*. |

To restart a running application, select the application you want to restart, click **Stop** and then click **Start**.

## Results

The status of the application changes and a message stating that the application started or stopped displays at the top the page.

## What to do next

You can configure an application so it does not start automatically when the server on which it resides starts. You then start the application manually using options described in this article.

If you want your application to start automatically when its server starts, you can adjust values that control how quickly the application or its server starts:

1. Go the settings page for your enterprise application. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Startup behavior**.
2. Specify a different value for **Startup order**.

   This setting specifies the order in which applications are started when the server starts. The default value is `1` in a range from 0 to 2147483647. The application with the lowest starting weight is started first.
3. Specify a different value for **Launch application before server completes startup**.

   This setting specifies whether the application must initialize fully before its server starts. The default value of `false` prevents the server from starting completely until the application starts. To reduce the amount of time it takes to start the server, you can set the value to `true` and have the application start on a background thread, thus allowing server startup to continue without waiting for the application.
4. Save the changes to the application configuration.

# Disabling automatic starting of applications

You can enable and disable the automatic starting of an application. By default, an installed application starts automatically when the server on which the application resides starts.

## Before you begin

This topic assumes that the application is installed on an application server and that the application starts automatically when the server starts.

This topic also assumes that you mapped the installed application to a server and that you have an administrative role with an authority higher than monitor.

### About this task

You might want an application to run only after you start it manually and not to run every time after the server starts. The target mapping for an application controls whether an application starts automatically when the server starts or requires you to start the application manually.

You must have an administrative role with an authority higher than monitor to change the automatic starting setting.

1. Go to the Target specific application status page for your application.

   Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Target specific application status**.
2. Select the target server on which the application resides.
3. Click **Disable Auto Start**.
4. Save changes to the administrative configuration.

### Results

The application does not start when its server starts. You must start the application manually.

### What to do next

To enable automatic starting of the application, do the following:

1. On the Target specific application status page for the application, select the target on which the application resides.
2. Click **Enable Auto Start**.
3. Save changes to the configuration.

## Target specific application status

Use this page to view mappings of deployed applications or modules to servers.

Also use this page to enable or disable the automatic starting of an application when the server on which the application resides starts.

To view this administrative console page, click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Target specific application status**.

When security is enabled, a separate application list is shown for each of your administrative roles. Supported roles include monitor, configurator, operator, administrator, deployer, and administrative security manager. For example, when you have the administrator role, the statement “You can administer the following resources” is shown followed by a list of servers that you can administer.

### Target

States the name of the target server to which the application or module maps. You specify the target on the Manage modules page accessed from the settings for an application.

### Node

Specifies the node name if the target is a server.

### Version

Specifies the version level of the target. The target can be a Version 7.x, 6.x or 5.x deployment target.

A *7.x deployment target* is a server with all members on a WebSphere Application Server Version 7.0 or later product.

A *6.x deployment target* is a server with all members on a WebSphere Application Server Version 6 product.

A *5.x deployment target* is a server with at least one member on a WebSphere Application Server Version 5 product.

An application, enterprise bean (EJB) module, Session Initiation Protocol (SIP) module (SAR), or Web module must be installed on a Version 7.x target under any of the following conditions:
- The module supports Java Platform, Enterprise Edition (Java EE) 5.
- The module calls a 7.x runtime application programming interface (API).
- The module uses a 7.x product feature.

If a module supports Java 2 Platform, Enterprise Edition (J2EE) 1.4, then you must install the module on a Version 6.x or 7.x deployment target. Modules that call a 6.1.x API or use a 6.1.x feature can be installed on a 6.1.x or 7.x deployment target. Modules that call a 6.0.x API or use a 6.0.x feature can be installed on a 6.0.x, 6.1.x or 7.x deployment target. Modules that require 6.1.x feature pack functionality can be installed on a 7.x deployment target or on a 6.1.x deployment target that has been enabled with that feature pack.

If JavaServer Pages (JSP) precompilation, EJB deployment (ejbdeploy), or Web Services deployment (wsdeploy) are enabled, then you can deploy applications to only those targets that have same product version as the deployment manager. If applications are targeted to servers that have an earlier version than the deployment manager, then you cannot deploy to those targets. Thus, if JSP precompilation, ejbdeploy, or wsdeploy are enabled, then you must deploy the application on a 6.1.x or 7.x target.

You can install an application or module developed for a Version 5.x product on a 7.x, 6.x or 5.x deployment target.

Similarly, a resource adapter (connector) module, or RAR file, developed for a Version 5.x product can reside on a 7.x, 6.x or 5.x target, provided the module does not support Java Cryptography Architecture (JCA) 1.5 and does not call any 7.x or 6.x runtime application programming interfaces (APIs). If the module supports JCA 1.5 or calls a 7.x or 6.x API, then the module must reside on a 7.x or 6.x target.

## Auto Start

Specifies whether the application modules installed on the target server are started (or enabled) when the server starts. This setting specifies the initial state of application modules. A **Yes** value indicates that the corresponding modules are enabled and thus are accessible when the server starts. A **No** value indicates that the corresponding modules are not enabled and thus are not accessible when the server starts.

By default, Auto Start is enabled. Thus, by default an installed application starts automatically when the server on which the application resides starts.

If you have an administrative role with an authority higher than monitor, you can enable and disable the automatic starting of the application. To disable the automatic starting of the application, enable the **Select** check box beside the target server and click **Disable Auto Start**. When automatic starting is disabled, the application does not start when its server starts. To enable the automatic starting of the application, select the target and click **Enable Auto Start**.

## Application Status

Indicates whether the application deployed on the application server is started, stopped, or unknown.

| | | |
|---|---|---|
| | **Started** | Application is running. |

| | | |
|---|---|---|
| | **Partial Start** | Application is in the process of changing from a *Stopped* state to a *Started* state. Application is starting to run but is not fully running yet. The application might be in the Partial Start state because one of its application servers is not started. |
| | **Stopped** | Application is not running. |
| | **Partial Stop** | Application is in the process of changing from a *Started* state to a *Stopped* state. Application has not stopped running yet. |
| | **Unknown** | Status cannot be determined.<br><br>An application with an unknown status might, in fact, be running but have an unknown status because the server running the administrative console cannot communicate with the server running the application. |
| | **Pending** | Status is temporarily unknown pending an event that a user did not initiate, such as pending an asynchronous call. |
| | **Not applicable** | Application does not provide information as to whether it is running. |

The status of an application on a Web server is always **Unknown**.

# Exporting enterprise applications

You can export an enterprise application to a location of your choice.

## About this task

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 product.

To export applications, use the **Export** button on the Enterprise applications page. Using **Export** produces an enhanced enterprise archive (EAR) file that contains the application as well as the deployment configuration. The *deployment configuration* consists of the deployment.xml and other configuration files that control the application behavior on a deployment target.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree to access the Enterprise applications page.
2. Select the check box beside the application and click **Export**.
3. On the Export application EAR files page, click on the link to download the exported EAR file.
4. Use the browser dialogue to specify a location at which to save the exported EAR file.

   User profile QEJBSVR must have *WX authority to the directory and at least *X authority to all directories in the path specified for the location.
5. Click **Back** to return to the Enterprise applications page.

## Results

The file containing binding information is exported to the specified node and directory, and has the name *enterprise_application_name*.ear.

Using the **Export** button to export applications does not export any manual changes that were made to applications in the installedApps directory. To export those changes, you must copy and move the application files manually.

## What to do next

You can edit your exported enhanced EAR file and then reinstall it. By default, installation expands an EAR file in the *profile_root*/installedApps/*cell_name* directory. If you specified the `$(CELL)` variable for **Directory to install application** on the Select installation options panel of the application installation wizard when you first installed the application, the *cell_name* directory is the current cell name.

To reinstall the enhanced EAR file, do either of the following:

- Use the **Update** operation available from the Enterprise applications page to upgrade the existing application installation.

  The **Update** operation adds the application files to the *profile_root*/installedApps/*cell_name* directory, where *cell_name* is the current cell name or the name of the cell that you specified for **Directory to install application** when you first installed the application on a deployment target. The **Directory to install application** setting is on the Select installation options panel of the application installation wizard. If you specified the `$(CELL)` variable for **Directory to install application** when you first installed the application, the *cell_name* directory is the current cell name.
- Use the **Applications** → **New application** → **New Enterprise Application** operation to install the exported EAR file.

  If you specified the `$(CELL)` variable for **Directory to install application** when you first installed the application, the *cell_name* directory is the current cell name. That is, if the file is originally installed on Cell1 with `$(CELL)` variable in the destination directory and you reinstall the enhanced EAR file on Cell2, the *cell_name* directory is Cell2, the current cell name.

  If the `$(CELL)` variable was not specified for the first installation, using **New Enterprise Application** to reinstall an enhanced EAR file installs the application in the *cell_name* directory of the exported application. That is, if the application is originally installed on and exported from Cell1 and you reinstall the enhanced EAR file on Cell2, the *cell_name* directory is Cell1. The enhanced EAR file expands in the Cell1 directory even though the current cell name is Cell2. By default, the application destination directory contains Cell1 in its path because the deployment.xml file in the exported application has Cell1 in it.

  If you exported the application from Cell1 and did not specify the `$(CELL)` variable when first installing the application, and you want to install the enhanced EAR file on a different cell, deselect **Process embedded configuration** on the Select installation options panel of the application installation wizard to expand the enhanced EAR file in the current cell name directory, which is not Cell1.

# Exporting enterprise application files

You can export individual files of a Java Platform, Enterprise Edition (Java EE) application or module.

## Before you begin

This topic assumes that you have installed an application or module on a server and that you want to export a file in the application or module.

## About this task

Exporting a file in a deployed application or module downloads the file to a location of your choice.

To export a file using the administrative console, use **Export File**.

To export an entire application, use **Export**. For information on **Export**, see “Exporting enterprise applications” on page 98. The exported enterprise archive (EAR) file contains application configuration data as well as the application.

1. Click **Applications** → **Application Types** → **WebSphere 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 File**. A drop-down list of exportable files is displayed.
3. Select a file from the list and click **Export**. A dialog in which you select a target location is displayed.

   If the browser does not prompt for a location to store the file, click **File** → **Save as** and specify a location to save the file that is shown in the browser.
4. Specify the location to which to download the file.

   User profile QEJBSVR must have *WX authority to the directory and at least *X authority to all directories in the path specified for the location.

### Results

The file is downloaded to the specified location.

### What to do next

Click **Back** to return to the Enterprise applications page.

## Exporting DDL files

You can export data definition language (DDL) files in the enterprise bean (EJB) modules of an application.

### About this task

Exporting DDL (Table.ddl) files in the EJB modules of an application downloads the DDL files to a location of your choice.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the administrative console navigation tree to access the Enterprise applications page.
2. Place a check mark in the check box beside the application and click **Export DDL**. If the application has no DDL files in any of its EJB modules, then the message *No DDL files were found* is displayed at the top of the page. If the application has DDL files in its EJB modules, then a page listing DDL files in the format `application_name.ear/_module.jar_Table.ddl` is displayed.
3. Click on a file in the list and specify the location to which to download the file.

   User profile QEJBSVR must have *WX authority to the directory and at least *X authority to all directories in the path specified for the location.

   **Note:** For Firefox browsers, right-click the file name, select **Save Link As**, and specify the location to which to download the file.

   Mozilla browsers might display the contents of the Table.ddl file instead of saving the file to disk. To save the file, edit the **Helper Application** preference settings of the Mozilla browser by adding a new type for DDL and specifying that you want to save DDL files to disk. That is, set `MIME type = ddl` and `Extension = ddl`.

### Results

The product downloads the DDL file to the specified location.

## Updating enterprise application files

You can update Java Platform, Enterprise Edition (Java EE) application files deployed on a server.

### Before you begin

Update your Java EE application or modules and reassemble them using an assembly tool. Typical tasks include adding or editing assembly properties, adding or importing modules into an application, and adding enterprise beans, Web components, and files.

Also, determine whether the updated files can be installed to your deployment targets. WebSphere Application Server Version 7.x and later supports Java EE 5 enterprise applications and modules.

If you are deploying Java EE 5 modules, ensure that the deployment target supports Version 7.x. You can deploy Java EE 5 modules to Version 7.x servers only. You cannot deploy Java EE 5 modules to Version 6.x deployment targets.

### About this task

Updating consists of adding a new file or module to an installed application, or replacing or removing an installed application, file or module. After replacement of a full application, the old application is uninstalled. After replacement of a module, file or partial application, the old installed module, file or partial application is removed from the installed application.

1. Determine which method to use to update your application files. The product provides several ways to update modules.
2. Update the application files using
   - Administrative console
   - wsadmin scripts
   - Java application programming interfaces
   - WebSphere rapid deployment of Java EE applications

   In some situations, you can update applications or modules without restarting the application server using hot deployment. Do not use hot deployment unless you are an experienced user and are updating applications in a development or test environment.
3. Start the deployed application files using
   - Administrative console
   - wsadmin startApplication
   - Java programs that use ApplicationManager or AppManagement MBeans

### What to do next

Save the changes to your administrative configuration.

Next, test the application. For example, point a Web browser at the URL for a deployed application (typically http://*hostname*:9060/*Web_module_name*, where *hostname* is your valid Web server and 9060 is the default port number) and examine the performance of the application. If the application does not perform as desired, edit the application configuration, then save and test it again.

## Ways to update enterprise application files

You can update Java Platform, Enterprise Edition (Java EE) application files deployed on a server in several ways.

*Table 4. Ways to update application files*

| Option | Method | Comments | Starting after update |
|---|---|---|---|
| Administrative console update wizard<br><br>See "Updating enterprise applications with the console" on page 103.<br><br>To remove a single file from a Java EE application or module, see "Removing enterprise files" on page 117. | Briefly, do the following:<br><br>1. Go to the Enterprise Applications page. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree.<br><br>2. Select the application to update and click **Update**.<br><br>3. On the Preparing for application update page, identify the application, module or files to update and click **Next**.<br><br>4. Complete steps in the update wizard and click **Finish**. | On the Preparing for application update page:<br><br>• Use **Full application** to update an .ear file.<br><br>• Use **Single module** to update a .war, .sar, enterprise bean .jar, or connector .rar file.<br><br>• Use **Single file** to update a file other than an .ear, .war, .sar, EJB .jar, or .rar file.<br><br>• Use **Partial application** to update or remove multiple files. | On the Enterprise applications page, select the updated application and click **Start**. |
| wsadmin scripts | Use the **update** command or the **updateInteractive** command in a script or at a command prompt. | ″Getting started with scripting″ in the Using the administrative clients PDF provides an overview of wsadmin. | Start the application using the **invoke** command and the startApplication attribute. |
| Java application programming interfaces | Update deployed applications by completing the steps in ″Managing applications through programming″ in the Using the administrative clients PDF. | Update an application in the following ways:<br><br>• Update the entire application<br><br>• Add to, replace or delete multiple files in an application<br><br>• Add a module to an application<br><br>• Update a module in an application<br><br>• Delete a module in an application<br><br>• Add a file to an application<br><br>• Update a file in an application<br><br>• Delete a file in an application | • Invoke the AdminApp *startApplication* command.<br><br>• Invoke the *startApplication* method on an ApplicationManager MBean using AdminControl. |
| Rapid deployment tools<br><br>See topics under **Rapid deployment of J2EE applications**. | Briefly, do the following:<br><br>1. Update your J2EE application files.<br><br>2. Set up the rapid deployment environment.<br><br>3. Create a free-form project.<br><br>4. Launch a rapid deployment session.<br><br>5. Drop your updated application files into the free-form project. | Rapid deployment tools offer the following advantages:<br><br>• You do not need to assemble your J2EE application files prior to deployment.<br><br>• You do not need to use other installation tools mentioned in this table to deploy the files. | Use any of the above options to start the application. Clicking **Start** on the Enterprise applications page is the easiest option. |

*Table 4. Ways to update application files (continued)*

| Option | Method | Comments | Starting after update |
| --- | --- | --- | --- |
| Hot deployment and dynamic reloading | Briefly, do the following:<br><br>1. Update your application (.ear), Web module (.war), enterprise bean .jar or HTTP plug-in configuration file.<br><br>2. Follow instructions in Hot deployment and dynamic reloading to update your file. | If you are new to WebSphere Application Server, use the administrative console to update applications. That option is easier.<br><br>Hot deployment and dynamic reloading is more difficult to complete. You must directly manipulate the application or module file on the server where the application is deployed. | Use any of the above options to start the application. Clicking **Start** on the Enterprise applications page is the easiest option. |

You can update .ear, enterprise bean .jar, Web module .war, Session Initiation Protocol (SIP) module (.sar), connector .rar, application client .jar, and any other files used by an installed application.

If the application is updated while it is running, WebSphere Application Server automatically stops the application, updates the application logic and restarts the application. If the application does not start automatically, start it manually using one of the **Starting** options. For more information on the restarting of updated applications, refer to "Fine-grained recycle behavior" in *IBM WebSphere Developer Technical Journal: System management for WebSphere Application Server V6 -- Part 5 Flexible options for updating deployed applications*.

## Updating enterprise applications with the console

Updating enterprise applications consists of adding a new file or module to an installed Java Platform, Enterprise Edition (Java EE) application, or replacing or removing an installed application, file or module.

### Before you begin

Before you update the application files on a server, ensure that the files are assembled in deployable modules.

Next, refer to "Ways to update enterprise application files" on page 101 and decide how to update your application files. You can update enterprise applications or modules using the administrative console, the wsadmin tool, or Java MBean programming. These ways provide similar updating capabilities.

Further, ensure that the updated files can be installed to your deployment targets.

### About this task

This topic describes how to update deployed applications or modules using the administrative console.

1. Back up the installed application or module.
   a. Go to the Enterprise applications page of the administrative console. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree.
   b. Export the application to an EAR file or export a file in the application. Select the application you want to export and click **Export** or **Export File**. Exporting preserves the binding information.
2. With the application selected on the Enterprise applications page, click **Update**. The Preparing for application update page is displayed.
3. Under **Specify the EAR, WAR, SAR or JAR module to upload and install**:
   a. Ensure that **Application to be updated** refers to the application to be updated.
   b. Under **Application update options**, select the installed application, module, or file that you want to update.

The online help Preparing for application update settings provides detailed information on the options.

**Note:** You cannot add, remove, or modify a Java Application Programming Interface (API) for XML-Based Web Services (JAX-WS) annotation using the **Replace or add a single file** or **Replace, add, or delete multiple files** update options. These options change a single file or a partial application. If you change a JAX-WS annotation using either of these options, the product does not return an error. However, you might encounter problems deploying annotated Web services.

4. If you selected the **Replace the entire application** or **Replace or add a single module** option:
   a. Click **Next** to display a wizard for updating application files.
   b. Complete the steps in the update wizard.

      This update wizard, which is similar to the installation wizard, provides fields for specifying or editing application binding information. Refer to information on installing applications and on the application installation binding settings page 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 Enterprise JavaBeans (EJB) Java Naming and Directory Interface (JNDI) names, EJB references or resource references, then those bindings will be part of the merged binding information. If new bindings are not present, then bindings are taken from the installed (old) version. If bindings are not present in the old version and if the default binding generation option is enabled, then the default bindings will be part of the merged binding information.

      You can select whether to ignore bindings in the old version or ones in the new version.
5. Click **Finish**.
6. If you did not use the Manage modules page of the update wizard, after updating the application, map the installed application or module to servers.

   Use the page accessed from the Enterprise Applications page.
   a. Go to the Manage modules page. Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Manage modules**.
   b. Specify the application server where you want to install modules contained in your application and click **OK**.

      You can deploy Java 2 Platform, Enterprise Edition (J2EE) 1.4 modules to servers on Version 6 or later nodes. You can deploy Java Platform, Enterprise Edition (Java EE) 5 modules to servers on Version 7.x nodes only.

## Results

After replacement of a full application, the product uninstalls the old application. After replacement of a module, file or partial application, the product removes the old installed module, file or partial application from the installed application.

## What to do next

After the application file or module installs successfully, do the following:

1. Save the changes to your configuration.
2. If needed, restart the application manually so the changes take effect.

   If the application is updated while it is running, the product automatically stops the application or only its changed components, updates the application logic, and restarts the stopped application or its components.
3. If the application you are updating is deployed on a server that has its application class loader policy set to `Single` on the application server settings page, restart the server.

# Preparing for application update settings

Use this page to update enterprise applications, modules or files already installed on a server.

To view this administrative console page, do the following:

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications**.
2. Select the installed application or module that you want to update.
3. Click **Update**.

Clicking **Update** displays a page that helps you update application files deployed in the cell. You can update the full application, a single module, a single file, or part of the application. If a new file or module has the same relative path as a file or module already existing on the server, the new file or module replaces the existing file or module. If the new file or module does not exist on the server, it is added to the deployed application.

## Application to be updated

Specifies the name of the installed (or deployed) application that you selected on the Enterprise applications page.

## Replace the entire application

Under **Application update options**, specifies to replace the application already installed on the server with a new (updated) enterprise application .ear file.

After selecting this option, do the following:

1. Specify whether the .ear file is on a local or remote file system and the full path name of the application. The path provides the location of the updated .ear file before installation.

   Use **Local file system** if the browser and the updated files or modules are on the same machine, whether or not the server is on that machine too. **Local file system** is available for all update options.

   Use **Remote file system** if the application file resides on any node in the current cell context.

   Also use the **Remote file system** option to specify an application file already residing on the machine running the application server. For example, the field value might be *app_server_install_root*/installableApps/test.ear. If you are installing a standalone WAR module, then specify the context root as well.

   **Note:** During application installation, application files typically are uploaded from a client machine running the browser to the server machine running the administrative console, where they are deployed. In such cases, use the Web browser running the administrative console to select modules to upload to the server machine. In some cases, however, the application files reside on the file system of any of the nodes in a cell. To have the application server install these files, use the **Remote file system** option.
2. If you are installing a standalone Web application (WAR) or a Session Initiation Protocol (SIP) module (SAR), specify the context root of the WAR or SAR file.

   The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is `/gettingstarted` and the servlet mapping is `MySession`, then the URL is http://host:port/gettingstarted/MySession.
3. Click **Next** to display a wizard for updating application files. The update wizard, which is similar to the installation wizard, provides fields for specifying or editing application binding information. Complete the steps in the update wizard as needed.

When the full application is updated, the old application is uninstalled and the new application is installed. When the configuration changes are saved and subsequently synchronized, the application files are expanded on the node where application will run. If the application is running on the node while it is updated, then the application is stopped, application files are updated, and application is started.

## Replace or add a single module

Under **Application update options**, specifies to replace a module in or add a module to an installed application.

The module can be a Web module (.war file), enterprise bean module (EJB .jar file), SIP module (.sar file), or resource adapter module (connector .rar file).

After selecting this option, specify whether the module is on a local or remote file system and the full path name of the module. The path provides the location of the updated module before installation. For information on **Local file system** and **Remote file system**, refer to the previous description of **Replace the entire application** .

To replace a module, the specified relative path (module URI) must match the path of the module to be updated in the installed application.

To add a new module to the installed application, the specified relative path must *not* match the path of a module in the installed application. The value specifies the desired path for the new module.

If you are installing a standalone Web or SIP module, specify a value for **Context root**. The context root is combined with the defined servlet mapping (from the .war file) to compose the full URL that users type to access the servlet. For example, if the context root is `/gettingstarted` and the servlet mapping is `MySession`, then the URL is http://host:port/gettingstarted/MySession.

Next, specify whether to show only installation options that require you to supply information or to show all installation options.

After specifying the required information on the module, click **Next** to display a wizard for updating application files. The update wizard, which is similar to the installation wizard, provides fields for specifying or editing module binding information. Complete the steps in the update wizard as needed.

After a single module is added or updated, when configuration changes are saved, the new or updated module is stored in the deployed application in the product configuration repository. When these changes are synchronized with the node, the module is added or updated to the node's file system. If the application is running on the node when the module is added or updated, then one of the following occurs:
- For updates to a Web module, the running Web module is stopped, Web module files are updated, and then the Web module is started.
- For module additions, the added module is started on the application servers where the application is running after it is expanded on the node. An application restart is not necessary.
- If the class loader policy for the application is set to `Single` so that all modules share a class loader, then the entire application is stopped and restarted for module level changes.
- If the security provider configured with the product does not support dynamic updates, then the entire application is stopped and restarted for module level changes.
- For all other updates to a module, the entire application is stopped, the module files are updated, then the entire application is started.

## Replace or add a single file

Under **Application update options**, specifies to replace a file in or add a file to an installed application.

Use this option to update a file used by the application that is not an .ear, .war, .sar, .rar or, in some instances, a .jar file. You can use this option to add or update .jar files that are not defined as modules in the application. To update an .ear, file use the **Replace the entire application** option. To update a .war file, .sar file, .rar file, or .jar file that is defined as a module in the application, use the **Replace or add a single module** option.

After selecting this option, specify whether the file is on a local or remote file system and the full path name of the file. The path provides the location of the updated file before installation. For information on **Local file system** and **Remote file system**, refer to the description of **Replace the entire application**.

For the relative path, specify a relative path to the file that starts from the root of the .ear file. For example, if the file is located at com/company/greeting.class in module hello.jar, specify a relative path of hello.jar/com/company/greeting.class.

To replace a file, the relative path must match the path of the file to be updated in the installed application.

To add a new file to the installed application, the must *not* match the path of a file in the installed application. The value specifies the desired path for the new file.

After you specify the file system and relative paths, click **Next**.

After a single file is added or updated, when configuration changes are saved, the new or updated file is stored in the deployed application in the product configuration repository. When these changes are synchronized with the node, the file is added or updated to the node's file system. If the application is running on the node when the file is added or updated, then one of the following occurs:
- When files are added at application metadata scope (META-INF directory) or updated at any application scope or in non-Web modules, the entire application is stopped, the file is added or updated, and then the entire application is restarted.
- When files are added at application non-metadata scope (outside of META-INF directory but not in any module), the changes are saved in the file system without restarting the running application.
- When files are added or updated to Web module metadata (META-INF or WEB-INF directory), the running Web module is stopped, the Web module file is added or updated, and then the Web module is started.
- For all other files in Web modules, the file is added or updated on the node's file system without stopping the application or any of its components.

### Replace, add, or delete multiple files

Under **Application update options**, specifies to update multiple files of an installed application by uploading a compressed file. Depending on the contents of the compressed file, a single use of this option can replace files in, add new files to, and delete files from the installed application. Each entry in the compressed file is treated as a single file and the path of the file from the root of the compressed file is treated as the relative path of the file in the installed application.

After selecting this option, specify whether the compressed file is on a local or remote file system and the full path name of the compressed file. You will likely use **Local file system** because you are uploading a compressed file and remote browsing only works for .ear, .sar, .war or .jar files. Specify a valid compressed file format such as .zip or .gzip. The path provides the location of the compressed file before installation. This option unzips the compressed file into the installed application directory.

Use **Local file system** if the browser and the updated files or modules are on the same machine, whether or not the server is on that machine too. **Local file system** is available for all update options.

To replace a file, a file in the compressed file must have the same relative path as the file to be updated in the installed application.

To add a new file to the installed application, a file in the compressed file must have a different relative path than the files in the installed application.

The relative path of a file in the installed application is formed by concatenation of the relative path of the module (if the file is inside a module) and the relative path of the file from the root of the module separated by /.

To remove a file from the installed application, specify metadata in the compressed file using a file named META-INF/ibm-partialapp-delete.props at any archive scope. The ibm-partialapp-delete.props file must be an ASCII file that lists files to be deleted in that archive with one entry for each line. The entry can contain a string pattern such as a regular expression that identifies multiple files. The file paths for the files to be deleted must be relative to the archive path that has the META-INF/ibm-partialapp-delete.props file.

| Level of files to delete | Metadata .props file to include in compressed file |
| --- | --- |
| Application | Include META-INF/ibm-partialapp-delete.props in the compressed file. In the metadata .props file, list files to be deleted. File paths are relative to the location of the META-INF/ibm-partialapp-delete.props file.<br><br>For example, to delete a file named utils/config.xmi from the root of the my.ear file, include the line utils/config.xmi in the META-INF/ibm-partialapp-delete.props file. |
| Module | Include *module_uri*/META-INF/ibm-partialapp-delete.props in the compressed file.<br><br>To delete one file from a module, include the file path relative to the module in the metadata .props file. For example, to delete `a/b/c.jsp` from the my.jar module, include `a/b/c.jsp` in my.jar/META-INF/ibm-partialapp-delete.props file in the compressed file.<br><br>To delete multiple files within a module, list the files to be deleted in the metadata .props file with one entry on each line. For example, to delete all JavaServer Pages (.jsp files) from the my.war file, include the line `.*jsp` in the my.war/META-INF/ibm-partialapp-delete.props file. The line uses a regular expression, `.*jsp`, to identify all .jsp files in my.war. |

You can use a single partial application file to add, delete and update multiple files.

After you specify a file system path, click **Next**.

After a partial application update, when configuration changes are saved, the new or updated application file is stored in the deployed application in the WebSphere Application Server configuration repository. When these changes are synchronized with the node, the files are added or updated to the node's file system. Because the partial application option updates multiple files, the application components that are restarted are determined using individual files in the partial application.

An example of entries in a partial application compressed file follows:

```
util.jar
META-INF/ibm-partialapp-delete.props
foo.jar/com/mycomp/xyz.class
xyz.war/welcome.jsp
xyz.war/WEB-INF/web.xml
webmod.war/META-INF/ibm-partialapp-delete.props
```

For this example, the META-INF/ibm-partialapp-delete.props file contains the `.*.dat` and tools/test.jar files. The webmod.war/META-INF/ibm-partialapp-delete.props file contains the `com/test/.*.jsp` and WEB-INF/test.xmi files.

The partial application update option does the following:
- Adds or replaces `util.jar` in the deployed application.
- Adds or replaces `com/mycomp/xyz.class` inside the foo.jar file of the deployed application.
- Deletes `*.dat` files from the application, but not from any modules.
- Deletes `tools/test.jar` from the application.
- Adds or replaces `welcome.jsp` inside the xyz.war module of the deployed application.
- Replaces `WEB-INF/web.xml` inside the xyz.war module of the deployed application.
- Deletes `com/test/*.jsp` from the webmod.war module.
- Deletes `WEB-INF/test.xmi` from the webmod.war module.

## Hot deployment and dynamic reloading

You can make various changes to applications and their modules without having to stop the server and start it again. Making these types of changes is known as *hot deployment and dynamic reloading*.

### Before you begin

This topic assumes that your application files are deployed on a server and you want to upgrade the files.

See "Ways to update enterprise application files" on page 101 and determine whether hot deployment is the appropriate way for you to update your application files. Other ways are easier and hot deployment is appropriate only for experienced users.

Do not use hot deployment if you intend to export your application, generate a plug-in based on the application configuration, or perform other application management in the future. Changes that you make to your application files using hot deployment are not recognized by administrative console or wsadmin application management functions. Those functions recognize only the application files that administrative programs such as the console or wsadmin present during application installation, update or other management functions. The application management functions do not recognize files changed by hot deployment.

### About this task

Hot deployment is the process of adding new components (such as WAR files, EJB Jar files, enterprise Java beans, servlets, and JSP files) to a running server without having to stop the application server process and start it again.

Dynamic reloading is the ability to change an existing component without needing to restart the server in order for the change to take effect. Dynamic reloading involves:
- Changes to the implementation of a component of an application, such as changing the implementation of a servlet
- Changes to the settings of the application, such as changing the deployment descriptor for a Web module

As opposed to the changes made to a deployed application described in "Updating enterprise application files" on page 100, changes made using hot deployment or dynamic reloading do not use the administrative console or a wsadmin scripting command. You must directly manipulate the application files on the server where the application is deployed.

If the application you are updating is deployed on a server that has its application class loader policy set to `Single`, you might not be able to dynamically reload your application. At minimum, you must restart the server after updating your application.

1. Locate your expanded application files.

   The application files are in the directory you specified when installing the application or, if you did not specify a custom target directory, are in the default target directory, *app_server_root*/`installedApps/`*`cell_name`*. Your EAR file, `${APP_INSTALL_ROOT}/`*`cell_name`*`/`*`application_name`*`.ear`, points to the target directory. The `variables.xml` file for the node defines ${APP_INSTALL_ROOT}.

   It is important to locate the expanded application files because, as part of installing applications, a WebSphere application server unjars portions of the EAR file onto the file system of the computer that will run the application. These expanded files are what the server looks at when running your application. If you cannot locate the expanded application files, look at the binariesURL attribute in the `deployment.xml` file for your application. The attribute designates the location the run time uses to find the application files.

   For the remainder of this information on hot deployment and dynamic reloading, *`application_root`* represents the root directory of the expanded application files.

2. Locate application metadata files. The metadata files include the deployment descriptors (`web.xml`, `application.xml`, `ejb-jar.xml`, and the like), the bindings files (`ibm-web-bnd.xmi`, `ibm-app-bnd.xmi`, and the like), and the extensions files (`ibm-web-ext.xmi`, `ibm-app-ext.xmi`, and the like).

   Metadata XML files for an application can be loaded from one of two locations. The metadata files can be loaded from the same location as the application binary files (such as *application_root*/META-INF) or they can be loaded from the WebSphere configuration tree, ${CONFIG_ROOT}/cells/*cell_name*/ applications /*application_EAR_name*/deployments/*application_name*/. The value of the useMetadataFromBinary flag specified during application installation controls which location is used. If specified, the metadata files are loaded from the same location as the application binary files. If not specified, the metadata files are loaded from the application deployment folder in the configuration tree.

   **Note:** You can have `useMetadataFromBinaries=true`, change an extracted copy of your application using hot deployment, and have the changes take effect at run time by following the procedure in this topic. However, changes that you make to your application files using hot deployment are not recognized by console or wsadmin application management functions. Those functions recognize only the original application files and not the files changed by hot deployment. Do not use hot deployment if you intend to export your application, generate a plug-in based on the application configuration, or perform other application management in the future. Hot deployment enables you to quickly change application files; it does not support the full management lifecycle of an application.

   For the remainder of this information, *metadata_root* represents the location of the metadata files for the specified application or module.

3. Optional: Examine the values specified for **Reload classes when application files are updated** and **Polling interval for updated files** on the settings page for your application's class loader.

   If reloading of classes is enabled and the polling interval is greater than zero (0), the application files are reloaded after the application is updated. For JavaServer Pages (JSP) files in a Web module, a Web container reloads JSP files only when the IBM extension jspReloadingEnabled in the jspAttributes of the ibm-web-ext.xmi file is set to `true`. You can set jspReloadingEnabled to `true` when editing your Web module's extended deployment descriptors in an assembly tool.

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 or stopping enterprise applications" on page 94 provides information on using the administrative console to start, stop, or restart an application.

## Results

The application files are updated on the server.

Because you directly manipulated the application files on the server, you might not be able to later use the administrative console or a wsadmin scripting command to work with the files. For example, if you try exporting a manually changed application using **Export** on an Enterprise applications console page, your manual changes to an application in the installedApps directory are not exported. To export those changes, you must copy and move the application files manually.

## 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.

## About this task

There are several changes that you can make to deployed application files without stopping the server and starting it again.

**Note:** See "Ways to update enterprise application files" on page 101 and determine whether hot deployment is the appropriate way for you to update your application files. Other ways are easier and hot deployment is appropriate only for experienced users. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server.

The following table lists the changes that you can make by manipulating an application file on the server where the application is deployed. The table also states whether you use hot deployment or dynamic reloading to make the changes.

| Change | Hot deployment | Dynamic reloading |
|---|---|---|
| Update an existing application on a running server by providing a new EAR file. | Yes | Yes |
| Add a new application to a running server. | Yes | No |
| Remove an existing application from a running server. | Yes | No |
| Change or add files to existing EJB or Web modules. | Yes | No |
| Change the `application.xml` file for an application. | Not applicable | Yes |
| Change the `ibm-app-ext.xmi` file for an application. | Not applicable | Yes |
| Change the `ibm-app-bnd.xmi` file for an application. | Not applicable | Yes |
| Change a non-module Jar file contained in the EAR file. | Yes | Yes |

- Update an existing application on a running server by 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.
- Add a new application to a running server.

  Install an application using the administrative console or the wsadmin `install` command.
- 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 use the `stopApplication` attribute of the AdminControl object with the wsadmin tool and then run the `uninstall` command.
- Change or add files to existing EJB or Web modules.
  1. Update the application files in the *application_root* location.
  2. Restart the application.

     Use the administrative console to restart the application. Or use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

     If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

## Changing or adding WAR files

You can change Web application archives (WAR files) on application servers without having to stop the server and start it again.

### About this task

There are several changes that you can make to WAR files without stopping the server and starting it again.

**Note:** See “Ways to update enterprise application files” on page 101 and determine whether hot deployment is the appropriate way for you to update your WAR files. Other ways are easier and hot deployment is appropriate only for experienced users. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server.

The following table lists the changes that you can make by manipulating a WAR file on the server where the application is deployed. The table also states whether you use hot deployment or dynamic reloading to make the changes.

| Change | Hot deployment | Dynamic reloading |
| --- | --- | --- |
| Change an existing JavaServer Pages (JSP) file. | Not applicable | Yes |
| Add a new JSP file to an existing application. | Yes | Yes |
| Change an existing servlet class by editing and recompiling. | Not applicable | Yes |
| Change a dependent class of an existing servlet class. | Not applicable | Yes |
| Add a new servlet using the Invoker (Serve Servlets by class name) facility or add a dependent class to an existing application. | Yes | Not applicable |
| Add a new servlet, including a new definition of the servlet in the `web.xml` deployment descriptor for the application. | Yes | Not applicable |
| Change the `web.xml` file of a WAR file. | Yes | Yes |
| Change the `ibm-web-ext.xmi` file of a WAR file. | Not applicable | Yes |
| Change the `ibm-web-bnd.xmi` file of a WAR file. | Not applicable | Yes |

- Change an existing JavaServer Pages (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.

- 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.

- Change an existing servlet class by 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

     If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

     If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.
- Add a new servlet using the Invoker (Serve Servlets by class name) facility or add 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

     If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.
- Add a new servlet, including a new definition of the servlet in the `web.xml` deployment descriptor for the application.
  1. Place the new `.class` file directly in the *application_root*/*module_name*/`WEB-INF/classes` directory. If the `.class` file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/`WEB-INF/lib`.

     You can edit the `web.xml` file in place or copy it into the *application_root*/*module_name*/`WEB-INF/classes` directory. The new `.class` file will not trigger a reloading of the application.
  2. Restart the application.

     Use the administrative console to restart the application. Or use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool. After the application restarts, the new servlet is available for service.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

- 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 no longer automatically detects changes to class files. Both of these changes disable the automatic reloading function. The only way to re-enable automatic reloading is to change the appropriate property and restart the application. See other task descriptions in this file for information on restarting an application.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

## Changing or adding EJB Jar files

You can change enterprise bean (EJB) Jar files on application servers without having to stop the server and start it again.

### About this task

There are several changes that you can make to EJB Jar files without stopping the server and starting it again.

**Note:** See "Ways to update enterprise application files" on page 101 and determine whether hot deployment is the appropriate way for you to update your EJB Jar files. Other ways are easier and hot deployment is appropriate only for experienced users. You can use the update wizard of the administrative console to make the changes without having to stop and restart the server.

The following table lists the changes that you can make to EJB Jar files by manipulating an EJB file on the server where the application is deployed. The table also states whether you use hot deployment or dynamic reloading to make the changes.

| Change | Hot deployment | Dynamic reloading |
|---|---|---|
| Change the `ejb-jar.xml` file of an EJB Jar file. | Not applicable | Yes |
| Change the `ibm-ejb-jar-ext.xmi` or `ibm-ejb-jar-bnd.xmi` file of an EJB Jar file. | Not applicable | Yes |
| Change the `Table.ddl` file for an EJB Jar file. | Not applicable | Not applicable |
| Change the `Map.mapxmi` or `Schema.dbxmi` file for an EJB Jar file. | Not applicable | Yes |
| Update the implementation class for an EJB file or a dependent class of the implementation class for an EJB file. | Not applicable | Yes |
| Update the Home/Remote interface class for an EJB file. | Not applicable | Yes |
| Add a new EJB file to an existing EJB Jar file. | Yes | Yes |

- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

- 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.

- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.
- 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 use the `startApplication` and `stopApplication` attributes of the AdminControl object with the wsadmin tool.

## Changing the HTTP plug-in configuration

You can change the HTTP plug-in configuration without having to stop the server and start it again.

### About this task

There are several change that you can make to the HTTP plug-in configuration without stopping the server and starting it again.

**Note:** See "Ways to update enterprise application files" on page 101 and determine whether hot deployment is the appropriate way for you to update your HTTP plug-in configuration. Other ways are easier and hot deployment is appropriate only for experienced users.

The following table lists the changes that you can make to the HTTP plug-in configuration. The table also states whether you use hot deployment or dynamic reloading to make the changes.

| Change | Hot deployment | Dynamic reloading |
|---|---|---|
| Change the `application.xml` file to change the context root of a Web application archive (WAR file). | Yes | No |
| Change the `web.xml` file to add, remove, or modify a servlet mapping. | Yes | Yes |
| 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. | Yes | Yes |

- Change the `application.xml` file to change the context root of a WAR file.
  1. Change the `application.xml` file.
  2. If the plug-in configuration property **Automatically propagate plug-in configuration file** is selected for this plug-in, it is automatically regenerated whenever the `application.xml` file changes.

     See documentation on the Web server plug-in properties for information on how to set this property. You can also run the `GenPluginCfg.bat/sh` script, or issue a wsadmin command to regenerate the plug-in configuration file.
- Change the `web.xml` file to add, remove, or modify a servlet mapping.
  1. Change the `web.xml` file.
  2. If the plug-in configuration property **Automatically propagate plug-in configuration file** is selected for this plug-in, it is automatically regenerated whenever the `web.xml` file changes.

     See documentation on the Web server plug-in properties for information on how to set this property. You can also run the `GenPluginCfg.bat/sh` script, or issue a wsadmin command to regenerate the plug-in configuration file.

     If the Web application has file serving enabled or has a servlet mapping of /, the plug-in configuration does not have to be regenerated. In all other cases a regeneration is required.
- 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.
  1. Change the `server.xml` file or the `virtualhost.xml` file.
  2. If the plug-in configuration property **Automatically propagate plug-in configuration file** is selected for this plug-in, it is automatically regenerated whenever the `server.xml` file changes.

     See documentation on the Web server plug-in properties for information on how to set this property. You can also run the `GenPluginCfg.bat/sh` script, or issue a wsadmin command to regenerate the plug-in configuration file.

## Uninstalling enterprise applications

After an application no longer is needed, you can uninstall it.

### About this task

Uninstalling an application deletes the application from the product configuration repository and deletes the application binaries from the file system of all nodes where the application modules are installed.

1. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the administrative console navigation tree to access the Enterprise applications page.
2. If you need to retain a copy of the application, back up the application.
   a. Select the application to uninstall.
   b. Click **Export**.

   The product exports the application to an enterprise application (.ear) file, preserving the binding information.

3. Uninstall the application.
   a. Select the application to uninstall.
   b. Click **Uninstall**.
   c. On the Uninstall application panel, click **OK**.
4. Save changes made to the administrative configuration.

## Results

On single-server products, application binaries are deleted after you save the changes.

# Removing enterprise files

After a file is no longer needed, you can remove the file from a Java Platform, Enterprise Edition (Java EE) application or module deployed on a server.

## About this task

Removing a file deletes the file from the product configuration repository and deletes the file from the file system of all nodes where the file is installed.

You can use the administrative console to remove a file from an application or module.

- Remove a file from an application.
  1. Go to the Enterprise applications page. Click **Applications** → **Application Types** → **WebSphere enterprise applications** in the console navigation tree.
  2. Select the application that contains a file you want removed.
  3. Click **Remove File**. The Remove a file page is displayed
  4. Select the URI of the file that you want removed from the application.
  5. Back up the application.

     Under **Export before removing file**, select the application name and then specify the location to which you want the file exported.
  6. Click **OK** to remove the file.
- Remove a file from a module.
  1. Go to the Manage modules page.

     Click **Applications** → **Application Types** → **WebSphere enterprise applications** → ***application_name*** → **Manage modules** in the console navigation tree.
  2. Select the module from which you want to delete a file.
  3. Click **Remove File**. The Remove a file from a module page is displayed.
  4. Select the URI of the file that you want removed from the module.
  5. Back up the application.

     Under **Export before removing file**, select the application name and then specify the location to which you want the file exported.
  6. Click **OK** to remove the file.

## Results

The file is exported to the designated location and removed from the application or module. The application or standalone Web module that had a file removed is restarted so the changes take effect.

### What to do next

Save the changes to your administrative configuration.

On single-server products, application binaries are deleted after you save the changes.

## Deploying and administering applications: Resources for learning

Use the following links to find relevant supplemental information about deploying and administering applications using the administrative console. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

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

View links to additional information about:
- "Programming model and decisions"
- "Programming instructions and examples"
- "Administration"

### Programming model and decisions

- Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, Second Edition, http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/
- Java EE Tutorials, http://java.sun.com/javaee/reference/tutorials/
- Recommended reading list: J2EE and WebSphere Application Server, http://www.ibm.com/developerworks/websphere/library/techarticles/0305_issw/recommendedreading.html
- Java EE 5: Power and productivity with less complexity – An overview of Java EE 5 features and developer-productivity enhancements, http://www.ibm.com/developerworks/java/library/j-jee5/index.html?ca=drs-
- Rational Application Developer V7 Programming Guide, SG24-7501-00, http://www.redbooks.ibm.com/abstracts/sg247501.html?Open
- *IBM WebSphere Developer Technical Journal:* The top Java EE best practices, http://www.ibm.com/developerworks/websphere/techjournal/0701_botzum/0701_botzum.html

### Programming instructions and examples

- *IBM WebSphere: Deployment and Advanced Configuration*, Roland Barcia, *et al.*, ISBN 0131468626 (Prentice Hall, 2004)
- *IBM WebSphere Developer Technical Journal:* Co-hosting multiple versions of J2EE applications, http://www.ibm.com/developerworks/websphere/techjournal/0405_poddar/0405_poddar.html
- Automated Deployment of Enterprise Application Updates: Part 1 - Basic concepts, http://websphere.sys-con.com/read/47889.htm

### Administration

- *IBM WebSphere Developer Technical Journal:* System management for WebSphere Application Server V6 -- Part 1 Overview of system management enhancements, http://www.ibm.com/developerworks/websphere/techjournal/0501_williamson/0501_williamson.html
- *IBM WebSphere Developer Technical Journal:* System management for WebSphere Application Server V6 -- Part 5: Flexible options for updating deployed applications, http://www.ibm.com/developerworks/websphere/techjournal/0510_apte/0510_apte.html
- *WebSphere Application Server V6.1:* System Management Configuration Handbook, SG24-7304-00, http://www.redbooks.ibm.com/abstracts/SG247304.html?Open

# Chapter 6. Managing applications through programming

Through Java MBean programming, you can install, update, and delete a Java Platform, Enterprise Edition (Java EE) application on a WebSphere Application Server deployment target.

## Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

For information on the restarting of updated applications, refer to Fine-grained recycle behavior in *IBM WebSphere Developer Technical Journal: System management for WebSphere Application Server V6 -- Part 5 Flexible options for updating deployed applications*.

Before you can install or change an application on a deployment target, you must first create or update your application and assemble it using an assembly tool.

## About this task

Besides installing, uninstalling, and updating applications through programming, you can additionally install, uninstall, and update Java EE applications through the administrative console or the wsadmin tool. All three ways provide identical updating capabilities.

1. Perform any or all of the following tasks to manage your Java EE applications through programming.
   - Access the application management function.

     This topic provides examples to access the application management functionality:
     - From WebSphere Application Server code
     - From outside WebSphere Application Server
     - When WebSphere Application Server is not running
   - Install an application.

     This topic provides an example for initially installing an application on a deployment target such as a server .
   - Uninstall an application.

     This topic provides an example for uninstalling an application that resides on a deployment target.
   - Manipulate additional attributes for a deployed application.

     This topic provides an example for manipulating attributes that are not exposed through the AppDeploymentTask object.
   - Share sessions for application management.

     This topic provides an example for saving application-specific updates for a deployed application to a session, and then to the configuration repository.
   - Update an application.

     This topic provides an example for updating the installed application on a server with a new application. When you completely update an application, the deployed application is uninstalled and the new enterprise archive (EAR) file is installed.
   - Add to, update, or delete part of an application.

     This topic provides an example that you can use to add, update, or delete part of an application on a server .
   - Edit an application.

     This topic provides an example that you can use to edit an application on a server .
   - Add a module.

© Copyright IBM Corp. 2008

This topic provides an example for adding a module to an application that resides on a server .

- Update a module.

  This topic provides an example for updating a module that resides on a server . When you update a module, the deployed module is uninstalled and the updated module is installed.
- Delete a module.

  This topic provides an example for deleting a module that resides on a server . When you delete a module, the deployed module is uninstalled.
- Add a file.

  This topic provides an example for adding a file to an application that resides on a server .
- Update a file.

  This topic provides an example for updating a file on a server . When you update a file, the deployed file is uninstalled and the updated file is installed.
- Delete a file.

  This topic provides an example for deleting a file on a server . When you delete a file, the deployed file is uninstalled.

2. Save your changes to the master configuration repository.

### What to do next

If you have further application updates, you can do the updates through programming, the administrative console, or the wsadmin tool.

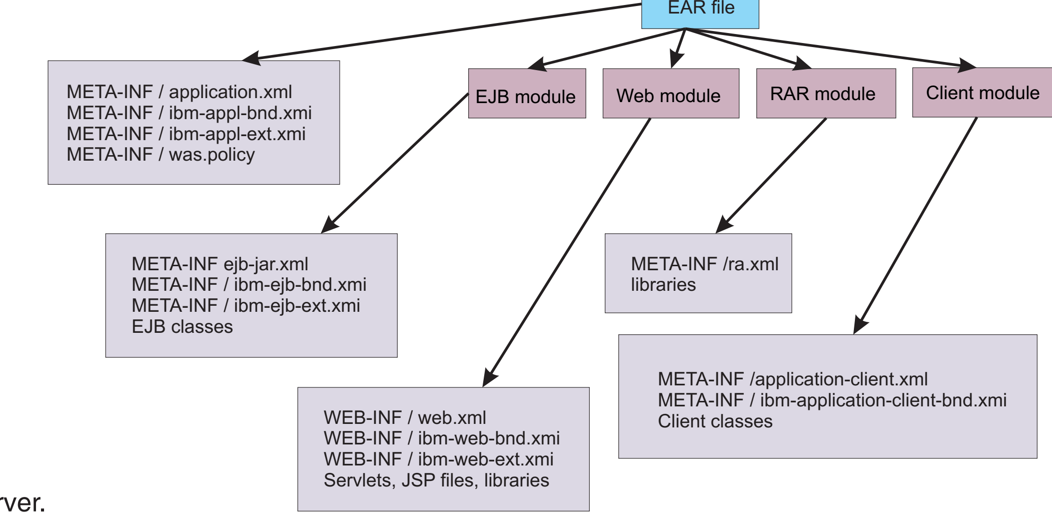
You can use the common deployment framework to add additional logic to application management operations. See “Extending application management operations through programming” on page 153. The tasks that the extensions provide are available through all the administrative clients, such as the wsadmin tool, the administrative console, or through programmatic APIs that the AppManagement MBean provides.

## Application management

Java 2 Platform, Enterprise Edition (J2EE) applications and modules include an Extensible Markup Language (XML)-based deployment descriptor that specifies various J2EE artifacts that pertain to applications or modules. The J2EE artifacts include Enterprise JavaBeans (EJB) definitions, security role definitions, EJB references, resource references, and so on. These artifacts define various unresolved references that the application logic uses. The J2EE specification requires that these artifacts map to J2EE platform-specific information, such as that found in WebSphere Application Server, during deployment of J2EE applications.

The application assembly tools that WebSphere Application Server supports, as well as the application management support that is provided with the product, facilitate collection of certain WebSphere Application Server information. The collected information is used to resolve references that are defined in various deployment descriptors in a J2EE application. This information is stored in the application EAR file in conjunction with the deployment descriptors. The following diagram shows the structure of an Enterprise

Archive (EAR) file that is populated with deployment information that is specific to WebSphere Application Server.




The application management architecture provides a set of classes with which deployers can collect WebSphere Application Server deployment information. This information is also referred to as *binding information*, and is stored in the application EAR file. The deployer can install the EAR file into a WebSphere Application Server configuration by using the AppManagement interface.

The application management support in WebSphere Application Server provides functions such as installing and uninstalling applications, editing binding information for installed applications, updating the entire application or part of the application, exporting the application, and so on. The com.ibm.websphere.management.application.AppManagement interface, which is exposed as a Java Management Extensions (JMX)-based AppManagement MBean in WebSphere Application Server, provides this functionality. Code that runs on the server or in a stand-alone administrative client program can access the interface. Access to the application management functions is also possible in the absence of WebSphere Application Server. This mode, known as *local mode*, is particularly useful for installing J2EE applications as part of product installation.

## Accessing the application management function

The com.ibm.websphere.management.application.AppManagementProxy class provides uniform access to application management functionality, regardless of whether the functionality is accessed from the server process, administrative client process, or a stand-alone Java program in the absence of WebSphere Application Server. This topic provides code excerpts that demonstrate how to obtain an AppManagementProxy instance in a variety of cases.

### Before you begin

This task assumes a basic familiarity with WebSphere Application Server programming interfaces and MBean programming. Read about WebSphere Application Server programming interfaces and MBean programming in the application programming interfaces documentation.

### About this task

Perform any of the following tasks to access application management functionality through programming.

- To access application management functionality from WebSphere Application Server code, for example, as a custom service, create the AppManagementProxy class.

```
AppManagement appMgmt =
              AppManagementProxy. getJMXProxyForServer();
```

- To access application management functionality from outside WebSphere Application Server through the AppManagement MBean, create an administrative client to establish a connection to WebSphere Application Server and then create the AppManagementProxy class.

  ```
  AdminClient adminClient = ....

  // create AppManagement proxy object
  AppManagement appMgmt =  AppManagementProxy. getJMXProxyForClient (adminClient);
  ```
- To access application management functionality when WebSphere Application Server is not running (local mode), create the AppManagementProxy class.

  ```
  AppManagement appMgmt = AppManagementProxy. getLocalProxy ();
  ```
  - When running in local mode set the com.ibm.ws.management.standalone system property to `true`. If you want to modify configuration documents in a non-default location, set the location of the configuration directory through the was.repository.root system property.
  - Although you can use application management functions with or without WebSphere Application Server running, do not access application management functions concurrently through local mode and the AppManagement MBean. Otherwise, updates that are made using these modes can collide and break the integrity of the WebSphere Application Server configuration.

### Results

After you successfully create the AppManagementProxy class, you have access to application management functionality.

### What to do next

You can perform various management tasks such as installing, uninstalling, editing, and so on.

## Installing an application through programming

You can install an application through the administrative console, the wsadmin tool, or programming. Use this example to install an application through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. Read about MBean programming in the application programming interfaces documentation.

Before you can install an application on WebSphere Application Server, you must first create or update your application and assemble it using an assembly tool.

### About this task

The AppDeploymentController instance contains meta-data defined in XMLl-based deployment descriptors as well as annotations defined in Java classes within the input enterprise archive (EAR) file.

Perform the following tasks to install an application through programming.

1. Populate the EAR file with WebSphere Application Server-specific binding information.
   a. Create the controller and populate the EAR file with appropriate options.
   b. Optionally run the default binding generator.
   c. Save and close the EAR file.
   d. Retrieve the saved options table that will be passed to the installApplication MBean API.
2. Connect to WebSphere Application Server.
3. Create the application management proxy.

4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the installApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.

      Refer to the com.ibm.websphere.management.application.AppManagement class in the application programming interfaces documentation to understand various options that can be passed to the installApplication MBean API.
5. Create the notification filter for listening to installation events.
6. Add the listener.
7. Install the application.
8. Wait for some timeout so that the program does not end.
9. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
10. When the installation is done, remove the listener and quit.

## Results

After you successfully run the code, the application is installed.

## Example

The following example shows how to install an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class Install {

    public static void main (String [] args) {

        try {
  String earFile = "C:/test/test.ear";
  String appName = "MyApp";

// Preparation phase: Begin
// Through the preparation phase you populate the enterprise archive (EAR) file with
// WebSphere Application Server-specific binding information. For example, you can specify
// Java Naming and Directory Interface (JNDI) names for enterprise beans, or virtual hosts
// for Web modules, and so on.

// First, create the controller and populate the EAR file with the appropriate options.
  Hashtable prefs = new Hashtable();
  prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());

// You can optionally run the default binding generator by using the following options.
// Refer to Java documentation for the AppDeploymentController class to see all the
// options that you can set.
  Properties defaultBnd = new Properties();
  prefs.put (AppConstants.APPDEPL_DFLTBNDG, defaultBnd);
  defaultBnd.put (AppConstants.APPDEPL_DFLTBNDG_VHOST, "default_host");

// Create the controller.
  AppDeploymentController controller = AppDeploymentController
    .readArchive(earFile, prefs);
  AppDeploymentTask task = controller.getFirstTask();
```

```
  while (task != null)
  {
// Populate the task data.
   String[][] data = task.getTaskData();
// Manipulate task data which is a table of stringtask.
   task.setTaskData(data);
   task = controller.getNextTask();
  }
  controller.saveAndClose();

  Hashtable options = controller.getAppDeploymentSavedResults();
// The previous options table contains the module-to-server relationship if it was set by
// using tasks.
//Preparation phase: End

// Get a connection to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

// Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

// If code for the preparation phase has been run, then you already have the options table.
// If not, create a new table and add the module-to-server relationship to it by uncommenting
// the next statement.
//Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());

// Uncomment the following statements to add the module to the server relationship table if
//  the preparation phase does not collect it.
//Hashtable module2server = new Hashtable();
//module2server.put ("*", target);
//options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, module2server);

//Create the notification filter for listening to installation events.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (AppConstants.NotificationType);


//Add the listener.
  NotificationListener listener = new AListener(_soapClient,
myFilter, "Install: " + appName, AppNotification.INSTALL);

// Install the application.
  proxy.installApplication (earFile, appName, options, null);
  System.out.println ("After install App is called..");

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }
```

```
}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit.

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

### What to do next

Once you install the application, you must explicitly start the application or you must stop and restart the server. For information on starting an application, see the Starting an application through programming topic in the *Using the administrative clients* PDF. For information on stopping or restarting the server, see the Stopping an application server topic or the Starting an application server topic, respectively, in the *Setting up the application serving environment* PDF.

## Starting an application through programming

You can start an application through the administrative console, the wsadmin tool, or programming. Use this example to start an application through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can start an application on WebSphere Application Server, you must first install your application.

### About this task

Perform the following tasks to start an application through programming.

1. Connect the administrative client to WebSphere Application Server.
2. Create the application management proxy.
3. Call the startApplication method on the proxy by passing the application name and optionally the list of targets on which to start the application.

### Results

After you successfully run the code, the application is started.

### Example

The following example shows how to start an application following the previously listed steps. Some statements are split on multiple lines for printing purposes.

```
//Do a get of the administrative client to connect to
//WebSphere Application Server.

AdminClient client = ...;
String appName = "myApp";
Hashtable prefs = new Hashtable();
// Use the AppManagement MBean to start and stop applications on all or some targets.

// The AppManagement MBean is on server1 in WebSphere Application Server.
// Query and get the AppManagement MBean.
ObjectName on = new ObjectName ("WebSphere:type=AppManagement,*");
Iterator iter = client.queryNames (on, null).iterator();
ObjectName appmgmtON = (ObjectName)iter.next();

//Start the application on all targets.
AppManagement proxy = AppManagementProxy.getJMXProxyForClient(client);
String started = proxy.startApplication(appName, prefs, null);
System.out.println("Application started on folloing servers: " + started);

//Start the application on some targets.
//String targets = "WebSphere:cell=cellname,node=nodename,
server=servername+WebSphere:cell=cellname,cluster=clusterName";
//String started1 = proxy.startApplication(appName, targets, prefs, null);
//System.out.println("Application started on following servers: " + started1)
```

## Uninstalling an application through programming

You can uninstall an application through the administrative console, the wsadmin tool, or programming. Use this example to uninstall an application through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can uninstall an application on WebSphere Application Server, you must first install it.

## About this task

Perform the following tasks to uninstall an application through programming.

1. Get a connection to WebSphere Application Server.
2. Get the application management proxy.
3. Create the notification filter for listening to uninstallation events.
4. Add the listener.
5. Uninstall the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the uninstallation is done, remove the listener and quit.

## Results

After you successfully run the code, the application is uninstalled.

## Example

The following example shows how to uninstall an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class Uninstall {

    public static void main (String [] args) {

        try {

// Get a connection to the server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Get the application management proxy.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());


  //Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
```

```
  myFilter.enableType (AppConstants.NotificationType);


  //Add the listener.
  NotificationListener listener = new AListener(_soapClient,
myFilter, "Install: " + appName, AppNotification.UNINSTALL);

  // Uninstall the application.
  proxy.uninstallApplication (appName, options, null);
  System.out.println ("After uninstall App is called..");

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the unistallation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
```

```
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
            }
        }
}
```

## Manipulating additional attributes for a deployed application

You can manipulate attributes for a deployed application through the administrative console, the wsadmin tool, or by programming. Use this example to manipulate attributes that are not exposed during or after application installation through the AppDeploymentTask object.

### Before you begin

This task assumes a basic familiarity with MBean programming and the ConfigService interfaces. Read about MBean programming and ConfigService interfaces in the application programming interfaces documentation.

### About this task

Perform the following tasks for your deployed application to manipulate attributes that are not exposed through the AppDeploymentTask object. The attributes are saved in the `deployment.xml` file that is created in the configuration repository for each deployed application.

1. Create a session.
2. Connect to WebSphere Application Server.
3. Locate the ApplicationDeployment object.
4. Manipulate the attributes.
5. Save your changes.
6. Clean up the session.

### Results

After you successfully run the code, the attributes are updated in the `deployment.xml` file for the deployed application.

### Example

The following example shows how to manipulate the startingWeight, warClassLoaderPolicy, and classloader attributes based on the previous steps.

```
import java.util.Properties;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.ObjectName;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.configservice.ConfigService;
import com.ibm.websphere.management.configservice.ConfigServiceHelper;
import com.ibm.websphere.management.configservice.ConfigServiceProxy;
import com.ibm.websphere.management.exception.ConfigServiceException;
import com.ibm.websphere.management.exception.ConnectorException;

public class AppManagementSample1 {

 public static void main(String[] args) {
```

```
String hostName = "localhost";
String port = "8880";
String appName = "ivtApp";

ConfigService configService = null;

// create a session.
Session session = new Session();

// establish connection to the server.
try {
 Properties props = new Properties();
 props.setProperty(AdminClient.CONNECTOR_TYPE,
   AdminClient.CONNECTOR_TYPE_SOAP);
 props.setProperty(AdminClient.CONNECTOR_HOST, hostName);
 props.setProperty(AdminClient.CONNECTOR_PORT, port);
 AdminClient adminClient =
  AdminClientFactory.createAdminClient(props);

 // create a config service proxy object.
 configService = new ConfigServiceProxy(adminClient);

 // Locate the application object.
 ObjectName rootID = configService.resolve(session,
   "Deployment="+appName)[0];
 System.out.println ("rootID is: " + rootID);

 // Locate the ApplicationDeployment object from the root.
 ObjectName appDeplPattern = ConfigServiceHelper.createObjectName
    (null, "ApplicationDeployment");
 /*
 ObjectName appDeplID = configService.queryConfigObjects(session,
   rootID, appDeplPattern, null)[0];
 */
 AttributeList list1 = configService.getAttributes(session,
   rootID, new String[]{"deployedObject"}, false);
 ObjectName appDeplID = (ObjectName)
  ConfigServiceHelper.getAttributeValue(list1, "deployedObject");
 System.out.println ("appDeplID: " + appDeplID);

 // Locate the class loader.

 // Change the starting weight through the startingWeight attribute. The starting weight
     // affects the order in which applications start.
 AttributeList attrList = new AttributeList();
 Integer newWeight = new Integer (10);
 attrList.add(new Attribute("startingWeight", newWeight));

 // Change the WAR class loader policy through the warClassLoaderPolicy attribute by
     // specifying SINGLE or MULTIPLE.
 // SINGLE=one classloader for all WAR modules
 attrList.add(new Attribute("warClassLoaderPolicy", "SINGLE"));

 // Set the class loader mode to PARENT_FIRST or PARENT_LAST.
 AttributeList clList = (AttributeList) configService.getAttribute
  (session, appDeplID, "classloader");
 ConfigServiceHelper.setAttributeValue (clList, "mode",
  "PARENT_LAST");
 attrList.add (new Attribute ("classloader", clList));

 // Set the new values.
 configService.setAttributes(session,  appDeplID, attrList);

 // Save your changes.
 configService.save(session, false);

} catch (Exception ex) {
```

```
    ex.printStackTrace();
   } finally {
            // Clean up the session.
    try {
     configService.discard(session);
    }
    catch (ConfigServiceException csEx)
    {
     csEx.printStackTrace();
    }
    catch (ConnectorException cnEx)
    {
     cnEx.printStackTrace();
    }
   }
  }
}
```

## Sharing sessions for application management

With the configuration service interface, ConfigService, you can create a session that is a temporary staging area, where you can save all the configuration modifications. Saving the session saves all the updates from the session into the WebSphere Application Server configuration repository. The application management logic supports session sharing with the configuration service. You can perform all the application management functions in the same session as the one that the configuration service creates. Saving such a session saves all the updates, including the ones that are application-specific.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

### About this task

Perform the following tasks for your deployed application to share and save application-specific updates through the configService configuration service.

1. Create a configuration service proxy object.
2. Create a session.
3. Pass the session information to the AppManagement MBean.

   Every method on the AppMangement interface takes session ID (workspace ID) as the last parameter. If the session information is passed in this parameter, the application management function uses the session. If you set the parameter to a null value:
   - No session sharing occurs
   - The configuration changes are always saved in the configuration repository if the operation succeeds.
4. Save the session after all the necessary changes are made.

   The following example outlines the general steps for session sharing through the configService configuration service. For a detailed example, see “Manipulating additional attributes for a deployed application” on page 129.

   ```
   public void installApplication (String localEarPath,
               String appName, Hashtable properties, String workspaceID)
               throws AdminException;

   AdminClient adminClient = ....;

   // Create a configuration service proxy object.
   ConfigService configService = new ConfigServiceProxy(adminClient);
   ```

```
// Create a session.
 Session session = new Session();

// Pass the session information to AppManagement MBean.
appMgmt = ...
appMgmt.installApplication
              (earPath, appName, properties, session.toString());
//Save the session after all necessary changes are made.
configService.save(session, false);
```

### Results

After you successfully complete the steps, you have saved application-specific updates for a deployed application to a session, and then to the configuration repository.

## Updating an application through programming

You can update an existing application through the administrative console, the wsadmin tool, or programming. Use this example to completely update an application through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update an application on WebSphere Application Server, you must first install your application.

### About this task

Perform the following tasks to completely update an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Prepare the enterprise archive (EAR) file by populating it with binding information.
6. Update the application.
7. Wait for some timeout so that the program does not end.
8. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
9. When the update is done, remove the listener and quit.

### Results

After you successfully run the code, the application is updated.

### Example

The following example shows how to update an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;
```

```
import javax.management.*;

public class aa {

    public static void main (String [] args) {

        try {

  // Connect to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
String fileContents = "/test/test.ear";
  // Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.INSTALL);

// Refer to the installation example to see how you can prepare the enterprise archive (EAR)
// file by populating it with binding information.
// If code for the preparation phase has started, then you already have the options table.
// If not, create a new table and add the module-to-server relationship to it by uncommenting
// the next statement.
//Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
  options.put ((AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_APP);

// Uncomment the following statements to add the module to the server relationship table if
//  the preparation phase does not collect it
//Hashtable module2server = new Hashtable();
//module2server.put ("*", target);
//options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, module2server);
// Update the application.
  proxy.updateApplication (   appName,
                null,
                fileContents,
                AppConstants.APPUPDATE_UPDATE,
                options,
                null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }
```

```
}

// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Adding to, updating, or deleting part of an application through programming

You can add to, update, or delete part of an existing application through the administrative console, the wsadmin tool, or programming. This example changes part of an application through programming. You can use this example whether you add to, update, or delete part of an existing application. Multiple changes to an application can be packaged in a single .zip file.

### Before you begin

To learn about the structure of the .zip file, see the Updating applications topic in the *Developing and deploying applications* PDF.

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add to, update, or delete part of an application on WebSphere Application Server, you must first install your application.

## About this task

Perform the following tasks to add to, update, or delete part of an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter.
4. Add the listener.
5. Partially change the existing application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the update is done, remove the listener and quit.

## Results

After you successfully run the code, you have changed the application.

## Example

The following example shows how to add to, update, or delete part of an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//partialApp specifies the location of the partial application.
//appName specifies the name of the application.
String partialApp = "/apps/partial.zip";
String appName = "MyApp";

//Do a get of the administrative client to connect to
//WebSphere Application Server.

AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

// Create the notification filter.
NotificationFilterSupport myFilter = new NotificationFilterSupport();
myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
//Add the listener.
NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);
//Partially change the existing application, MyApp.

Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_PARTIALAPP);

proxy.updateApplication ( appName,
    null,
    partialApp,
    null,
    options,
    null);
```

```
// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Editing applications

You can edit deployed applications through the administrative console, the wsadmin tool, or by programming. Use this example to edit a deployed application through programming.

## Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming, see MBean Java application programming interface (API) documentation.

Before you can edit an application on WebSphere Application Server, you must first install the application.

## About this task

Perform the following tasks to edit your deployed application.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Get information about an installed application.
4. Manipulate task data as necessary.
5. Save changes for the installed application.

## Results

After you successfully run the code, the application is edited.

## Example

The following example shows how to edit an application, based on the previous steps.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class aa {

    public static void main (String [] args) {

        try {

  // Connect to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

  // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
// Get information for an application with name appName:
// Pass Locale information as the preference.
Hashtable prefs = new Hashtable();
 prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());
Vector allTasks =  appMgmt.getApplicationInfo (appName, prefs, null);
```

```
// Manipulate task data as necessary.
if (task.getName().equals ("MapRolesToUsers") && !task. isTaskDisabled())
{
       // find out column index for role and user column
       // refer to the previous table to find the column names
       int roleColumn = -1;
       int userColumn = -1;
       String[] colNames = task.getColumnNames();
       for (int i=0; i < colNames.length; i++)
       {
            if (colNames[i].equals ("role"))
             roleColumn = i;
            else if (colNames[i].equals ("role.user"))
               userColumn = i;
       }

       // iterate over task data starting at row 1 as row0 is
       // column names
       String[][]data = task.getTaskData();
       for (int i=1; i < data.length; i++)
       {
            if (data[i][roleColumn].equals ("Role1"))
            {
             data[i][userColumn]="User1|User2";
               break;
            }
       }

       // now that the task data is changed, save it back
       task.setTaskData (data);
}

// Save changes back into the installed application:
// Set information for an application with name appName.
// Pass Locale information as the preference.
prefs = new Hashtable();
prefs.put(AppConstants.APPDEPL_LOCALE, Locale.getDefault());
appMgmt.setApplicationInfo (appName, prefs, null, allTasks);

        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

## Preparing a module and adding it to an existing application through programming

You can add a module to an existing application through the administrative console, the wsadmin tool, or programming. Use this example to add a module through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add a module to an application on WebSphere Application Server, you must install the application.

## About this task

Perform the following tasks to add a module to an application through programming.

1. Create an application deployment controller instance to populate the module file with binding information.
2. Save the binding information in the module.
3. Get the installation options.
4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the updateApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.
5. Connect to WebSphere Application Server.
6. Create the application management proxy.
7. Create the notification filter.
8. Add the listener.
9. Add the module to the application.
10. Specify the target for the new module.
11. Wait for some timeout so that the program does not end.
12. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
13. When the module addition is done, remove the listener and quit.

## Results

After you successfully run the code, the module is added to the application.

## Example

The following example shows how to add a module to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//moduleName specifies the name of the module that you add to the application.
//moduleURI specifies a URI that gives the target location of the module
// archive contents on a file system. The URI provides the location of the new
// module after installation. The URI is relative to the application URL.
//uniquemoduleURI specfies the URI that gives the target location of the
// deployment descriptor file. The URI is relative to the application URL.
//target specifies the cell, node, and server on which the module is installed.

String moduleName = "/apps/foo.jar";

String moduleURI = "Increment.jar";
String uniquemoduleURI = "Increment.jar+META-INF/ejb-jar.xml";
String target = "WebSphere:cell=cellname,node=nodename,server=servername";

//Create an application deployment controller instance, AppDeploymentController,
//to populate the Java Archive (JAR) file with binding information.
//The binding information is WebSphere Application Server-specific deployment information.

Hashtable preferences = new Hashtable();
preferences.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
preferences.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);
AppDeploymentController controller = AppManagementFactory.readArchiveForUpdate(
    moduleName,
    moduleURI,
    AppConstants.APPUPDATE_ADD,
    preferences,
    null);
```

If the module that you add to the application lacks any bindings, add the bindings so that the module addition works. Collect and add the bindings by using the public APIs provided with WebSphere Application Server. Refer to Java documentation for the com.ibm.websphere.management.application.client.AppDeploymentController instance to learn more about how to collect and populate tasks with WebSphere Application Server-specific binding information. The AppDeploymentController instance contains meta-data defined in XML-based deployment descriptors as well as annotations defined in Java classes within the input module.

```
//After you collect all the binding information, save it in the module.
controller.saveAndClose();

//Get the installation options.
Hashtable options = controller. getAppDeploymentSavedResults();

//Connect the administrative client, AdminClient, to WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Update the existing application, MyApp, by adding the module.
String appName = "MyApp";

options.put (AppConstants.APPUPDATE_CONTENTTYPE,
   AppConstants. APPUPDATE_CONTENT_MODULEFILE);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);



//Specify the target for the new module.
Hashtable mod2svr = new Hashtable();
options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, mod2svr);
mod2svr.put (uniquemoduleURI, target);
proxy.updateApplication ( appName,
    moduleURI,
    moduleName,
    AppConstants.APPUPDATE_ADD,
    options,
    null);

// Wait for some timeout. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;
```

```
    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Preparing and updating a module through programming

You can update a module for an existing application through the administrative console, the wsadmin tool, or programming. When you update a module, you replace the existing module with a new version. Use this example to update a module through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update a module on WebSphere Application Server, you must first install the application.

### About this task

Perform the following tasks to update a module through programming.

1. Create an application deployment controller instance to populate the Java archive file with binding information.
2. Save the binding information in the module.
3. Get the installation options.

4. If the preparation phase (population of the EAR file) is not performed, the do the following actions:
   a. Create an options table to be passed to the updateApplication MBean API.
   b. Create a table for module to server relations and add the table to the options table.
5. Connect to WebSphere Application Server.
6. Create the application management proxy.
7. Create the notification filter.
8. Add the listener.
9. Replace the module in the application.
10. Specify the target for the new module.
11. Wait for some timeout so that the program does not end.
12. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
13. When the module addition is done, remove the listener and quit.

## Results

After you successfully run the code, the existing module is replaced with the new one.

## Example

The following example shows how to add a module to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//moduleName specifies the name of the module that you add to the application.
//moduleURI specifies a URI that gives the target location of the module
// archive contents on a file system. The URI provides the location of the new
// module after installation. The URI is relative to the application URL.
//uniquemoduleURI specfies the URI that gives the target location of the
// deployment descriptor file. The URI is relative to the application URL.
//target specifies the cell, node, and server on which the module is installed.
//appName specifies the name of the application to update.

String moduleName = "/apps/foo.jar";

String moduleURI = "Increment.jar";
String uniquemoduleURI = "Increment.jar+META-INF/ejb-jar.xml";
String target = "WebSphere:cell=cellname,node=nodename,server=servername";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

Vector tasks = proxy.getApplicationInfo (appName, new Hashtable(), null);

//Create an application deployment controller instance, AppDeploymentController,
//to populate the Java archive (JAR) file with binding information.
//The binding information is WebSphere Application Server-specific deployment information.

Hashtable preferences = new Hashtable();
preferences.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
preferences.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);
AppDeploymentController controller = AppManagementFactory.readArchiveForUpdate(
    moduleName,
    moduleURI,
    AppConstants.APPUPDATE_UPDATE,
    preferences,
    tasks);
```

If the module that you update for the application lacks any bindings, add the bindings so that the module update works. Collect and add the bindings by using the public APIs that are provided with WebSphere Application Server. Refer to Java documentation for the AppDeploymentController instance to learn more about how to collect and populate tasks with WebSphere Application Server-specific binding information. The AppDeploymentController instance contains meta-data defined in XML-based deployment descriptors as well as annotations defined in Java classes within the input module.

```
//After you collect all the binding information, save it in the module.
controller.saveAndClose();

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);


//Get the installation options.
Hashtable options = controller. getAppDeploymentSavedResults();

//Update the existing application by adding the module.

options.put (AppConstants.APPUPDATE_CONTENTTYPE,
   AppConstants. APPUPDATE_CONTENT_MODULEFILE);

//Specify the target for the new module
Hashtable mod2svr = new Hashtable();
options.put (AppConstants.APPDEPL_MODULE_TO_SERVER, mod2svr);
mod2svr.put (uniquemoduleURI, target);

proxy.updateApplication ( appName,
    moduleURI,
    moduleName,
    AppConstants.APPUPDATE_UPDATE,
    options,
    null);
// Wait. The installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
```

```
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Deleting a module through programming

You can delete a module from an existing application through the administrative console, the wsadmin tool, or programming. Use this example to delete a module through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can delete a module from an application on WebSphere Application Server, you must first install the application.

### About this task

Perform the following tasks to delete a module through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Delete the module.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the module is deleted, remove the listener and quit.

### Results

After you successfully run the code, the existing module is deleted from the application.

## Example

The following example shows how to delete a module from an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//moduleURI specifies a URI that gives the target location of the module.
//appName specifies the name of the application to update.
String moduleURI = "Increment.jar";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.

AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

//Update the existing application, MyApp, by deleting the module.
Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_MODULEFILE);

proxy.updateApplication ( appName,
    moduleURI,
    null,
    AppConstants.APPUPDATE_DELETE,
    options,
    null);

// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
```

```
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Adding a file through programming

You can add a file to an existing application through the administrative console, the wsadmin tool, or programming. This example describes how to add a file through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can add a file to an application on WebSphere Application Server, you must first install the application.

### About this task

Perform the following tasks to add a file to an application through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Add the file to the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the file is added to the application, remove the listener and quit.

### Results

After you successfully run the code, the file is added to the application.

## Example

The following example shows how to add a file to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
import java.lang.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import com.ibm.websphere.management.application.*;
import com.ibm.websphere.management.application.client.*;
import com.ibm.websphere.management.*;

import javax.management.*;

public class FileAdd {

    public static void main (String [] args) {

        try {

// Get a connection to WebSphere Application Server.
  String host = "localhost";
  String port = "8880";
  String target = "WebSphere:cell=cellName,node=nodeName,server=server1";

  Properties config = new Properties();
  config.put (AdminClient.CONNECTOR_HOST,  host);
  config.put (AdminClient.CONNECTOR_PORT,  port);
  config.put (AdminClient.CONNECTOR_TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
  System.out.println ("Config: " + config);
     AdminClient _soapClient = AdminClientFactory.createAdminClient(config);

 // Create the application management proxy, AppManagement.
  AppManagement proxy = AppManagementProxy. getJMXProxyForClient (_soapClient);

  String appName = "MyApp";
  String fileURI = "test.war/com/acme/abc.jsp";
  String fileContents = "/temp/abc.jsp";
  //Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);

  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

  Hashtable options = new Hashtable();
  options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
  options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

  // Update the application
  proxy.updateApplication (   appName,
               fileURI,
               fileContents,
               AppConstants.APPUPDATE_ADD,
               options,
               null);

// Wait; the installation Application Programming Interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(90000); // Wait so that the program does not end.

        }
        catch (Exception e) {
```

```
            e.printStackTrace();
        }

    }

}


// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Updating a file through programming

You can update a file for an existing application through the administrative console, the wsadmin tool, or programming. This example describes how to update a file through programming.

## Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can update a file for an application on WebSphere Application Server, you must first install the application.

## About this task

Perform the following tasks to update a file through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Update the file in the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the installation is done, remove the listener and quit.

## Results

After you successfully run the code, the file is updated for the application.

## Example

The following example shows how to add a file to an application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//fileContents specifies the name of the file that you add to the application.
//appName specifies the name of the application.
//fileURI specifies a URI that gives the target location of the file. The URI
// provides the location of the new module after installation. The URI is
// relative to the application URL.
String fileContents = "/apps/test.jsp";
String appName = "MyApp";
String fileURI = "SomeWebMod.war/com/foo/abc.jsp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

Hashtable options = new Hashtable();
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

proxy.updateApplication ( appName,
    fileURI,
```

```
    fileContents,
    AppConstants.APPUPDATE_UPDATE,
    options,
    null);

// Wait; the installation application programming interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //When the installation is done, remove the listener and quit.

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
                    System.exit (0);
        }
    }
}
```

## Deleting a file through programming

You can delete a file from an existing application through the administrative console, the wsadmin tool, or programming. Use this example to delete a file through programming.

### Before you begin

This task assumes a basic familiarity with MBean programming. For information on MBean programming see MBean Java application programming interface (API) documentation.

Before you can delete a file from an application on WebSphere Application Server, you must first install the application.

### About this task

Perform the following tasks to delete a file through programming.

1. Connect to WebSphere Application Server.
2. Create the application management proxy.
3. Create the notification filter for listening to events.
4. Add the listener.
5. Delete the file from the application.
6. Wait for some timeout so that the program does not end.
7. Listen to Java Management Extensions (JMX) notifications to understand completion of the operation.
8. When the file is deleted from the application, remove the listener and quit.

### Results

After you successfully run the code, the file is deleted from the application.

### Example

The following example shows how to delete a file based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
//Inputs:
//fileURI specifies a URI that gives the target location of the file. The URI
// provides the location of the new module after installation. The URI is
// relative to the application URL.
//appName specifies the name of the application.

String fileURI = "Increment.jar/com/acme/Foo.class";
String appName = "MyApp";

//Get the administrative client to connect to
//WebSphere Application Server.
AdminClient client = ...;

//Create the application management proxy.
AppManagement proxy = AppManagementProxy. getJMXProxyForClient (client);

//Create the notification filter.
  NotificationFilterSupport myFilter = new NotificationFilterSupport();
  myFilter.enableType (NotificationConstants.TYPE_APPMANAGEMENT);
  //Add the listener.
  NotificationListener listener = new AListener(_soapClient, myFilter,
"Install: " + appName, AppNotification.UPDATE);

//Update the existing application, MyApp, by deleting the file.
Hashtable options = new Hashtable();
```

```
options.put (AppConstants.APPDEPL_LOCALE, Locale.getDefault());
options.put (AppConstants.APPUPDATE_CONTENTTYPE, AppConstants.APPUPDATE_CONTENT_FILE);

proxy.updateApplication ( appName,
    fileURI,
    null,
    AppConstants.APPUPDATE_DELETE,
    options,
    null);

// Wait for some timeout. The installation Application Programming Interface (API) is
//  asynchronous and so returns immediately.
// If the program does not wait here, the program ends.
  Thread.sleep(300000); // Wait so that the program does not end.
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }

}
// Specify the Java Management Extensions (JMX) notification listener for JMX events.
class AListener implements NotificationListener
{
    AdminClient _soapClient;
    NotificationFilterSupport  myFilter;
    Object handback;
    ObjectName on;
    String eventTypeToCheck;

    public AListener(AdminClient cl, NotificationFilterSupport fl,
Object h, String eType) throws Exception
    {
        _soapClient = cl;
        myFilter = fl;
        handback = h;
        eventTypeToCheck = eType;

        Iterator iter = _soapClient.queryNames (new ObjectName(
"WebSphere:type=AppManagement,*"), null).iterator();
        on = (ObjectName)iter.next();
        System.out.println ("ObjectName: " + on);
        _soapClient.addNotificationListener (on, this, myFilter, handback);
    }

    public void handleNotification (Notification notf, Object handback)
    {
            AppNotification ev = (AppNotification) notf.getUserData();
            System.out.println ("!! JMX event Recd: (handback obj= " + handback+ "): " + ev);


            //Once the installation is done, remove the listener and quit

            if (ev.taskName.equals (eventTypeToCheck) &&
                (ev.taskStatus.equals (AppNotification.STATUS_COMPLETED) ||
                 ev.taskStatus.equals (AppNotification.STATUS_FAILED)))
            {
                    try
                    {
                            _soapClient.removeNotificationListener (on, this);
                    }
            catch (Throwable th)
                    {
                        System.out.println ("Error removing listener: " + th);
                    }
```

```
                    System.exit (0);
            }
        }
}
```

## Extending application management operations through programming

You can use the common deployment framework to add additional logic to application management operations. The additional logic can do such tasks as code generation, configuration operations, additional validation, and so on. This topic demonstrates, through programming, how to plug into the *common deployment framework* to extend application management operations.

### Before you begin

This task assumes a basic familiarity with Java application programming interfaces (APIs). Read about the Java APIs in the application programming interfaces documentation.

Before you can extend application management operations, you must first install WebSphere Application Server.

### About this task

Use this example to extend application management through programming. The tasks that the extensions provide are available through all the administrative clients, such as the wsadmin tool, the administrative console, or through programmatic APIs that the AppManagement MBean provides.

1. Define your extension as an Eclipse plug-in and add a `plugin.xml` file to register your extension provider with the deployment framework.
   a. In the `plugin.xml` file, provide an extension provider implementation class for the common-deployment-framework-extensionprovider extension point.
   b. Put the plug-in Java archive (JAR) file in the `plugins` directory of your WebSphere Application Server installation.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.ibm.myproduct.MyExtensionProvider"
   name="My Extension"
   version="1.0.0">

   <extension point="common-deployment-framework-extensionprovider">
      <action class="com.acme.MyExtendProviderImpl"/>
   </extension>
</plugin>
```

2. Provide an extension provider.

   An extension provider class provides steps for a given operation on an application Enterprise archive (EAR) file. Before an operation runs, the deployment framework queries all the registered extension providers for additional steps. A single list of steps is passed to each provider. Each provider can add steps to the list. The default provider that the deployment framework provides is called first to populate the list with default steps. Other extension providers are called next.

   Various operations that you can extend through the common deployment framework are defined as constants in the DeploymentConstants class. These operations are described in the following table. Some operations are split on multiple lines for printing purposes.

*Table 5. Extensible DeploymentConstants operations*

| Operation | Description |
| --- | --- |
| DeploymentConstants.CDF_OP_INSTALLJ2EE | Installs a Java Platform, Enterprise Edition (Java EE) EAR file |

*Table 5. Extensible DeploymentConstants operations (continued)*

| Operation | Description |
| --- | --- |
| DeploymentConstants.CDF_OP_EDITJ2EE | Edits a deployment application configuration |
| DeploymentConstants.CDF_OP_UPDATEJ2EE | Applies a fine-grained update to an application such as addition, removal, or update of a file or a module; or partial update of an application |
| DeploymentConstants.CDF_OP_UNINSTALLJ2EE | Uninstalls a Java EE application |
| `DeploymentConstants.CDF_OP_CREATE_EAR_WRAPPERJ2EE` | Wraps the contents input to the application installation into an EAR file |

The AppManagement MBean, which is responsible for deploying and managing Java EE applications on WebSphere Application Server, runs all the operations except the CDF_OP_CREATE_EAR_WRAPPERJ2EE operation. Deploy the extensions that extend these operations in the `plugins` directory of the stand-alone Application Server .

Either the wsadmin utlity or the administrative console runs the CDF_OP_CREATE_EAR_WRAPPERJ2EE operation when the input contents that are supplied to the CDF_OP_INSTALLJ2EE operation are not packaged as an EAR file. Deploy an extension that extends the CDF_OP_CREATE_EAR_WRAPPERJ2EE operation in the `plugins` directory of the wsadmin installation.

The following example provides an extension provider that does the following tasks:

a. Adds two additional steps for the application installation operation

b. Adds one step for wrapping input contents into an EAR file

```
package com.acme;

import com.ibm.websphere.management.deployment.registry.ExtensionProvider;
import com.ibm.websphere.management.deployment.core.DeploymentConstants;

public class MyExtensionProviderImpl extends ExtensionProvider {
  public void addSteps (String type, String op, String phase,
   List steps)
  {
   if (op.equals (DeploymentConstants.CDF_OP_INSTALLJ2EE))
   {
    // Add a code generation step.
    steps.add (0, new com.acme.CodeGenStep());
    // Add a configuration step.
    steps.add (new com.acme.ConfigStep());
   }
   else if (op.equals (DeploymentConstants.CDF_OP_CREATE_EAR_WRAPPERJ2EE))
   {
    // Add an ear-wrapper step.
    steps.add (new com.acme.EarWrapperStep());
   }
  }
}
```

3. Provide the deployment step implementation.

An extension provider adds a deployment step. The step contains logic that performs additional processing in an application management operation. The logic provides the step access to the deployment context and the deployable object. The deployment context provides information, such as the name of the operation, the configuration session ID, a temporary location for creating temporary files, operation parameters, and so on. The deployable object wraps the deployment content input to the operation. For example, the deployable object wraps the Java EE EAR file for the installation operation or a file, a module, or a partial application for the update operation.

- The following example illustrates how an extension during installation entirely changes an EAR file that is input to the installation operation. The example provides a deployment step during the installation operation that does the following tasks:

a. Runs code generation to generate a new EAR file.
b. Calls the setContentPath method in the DeployableObject class to set the new EAR file path. The default installation logic, such as steps that the default installation logic adds, uses this new EAR file for installation in the configuration repository.

```
package com.acme;

import com.ibm.websphere.management.deployment.core.DeploymentStep;
import com.ibm.websphere.management.deployment.core.DeployableObject;

public class CodeGenStep extends DeploymentStep
{
  public void execute (DeployableObject dObject)
  {
   EARFile earFile = (EARFile)dObject.getHandle();
   String newEARPath = null;
   // Use step specific logic to create another EAR file after code generation.
   ...
   newEARPath = _context.getTempDir() + "new.ear";

   dObject.setContentPath (newEARPath);
  }
}
```

- The following example provides a deployment step that:
  a. Reads the contents of the input EAR file.
  b. Manipulates the configuration session accessed through the context instance, _context.

```
package com.acme;

public class ConfigStep extends DeploymentStep
{
  public void execute (DeployableObject dObject)
  {
   EARFile earFile = (EARFile) dObject.getHandle();

   // Use the following example code to perform the configuration.
   String sessionID = _context.getSessionID();
   com.ibm.websphere.management.Session session = new
    com.ibm.websphere.management.Session (sessionID, true);
   // Use the configuration service to perform the configuration steps.
   ...

   // Read the application configuration.
   Application appDD = earFile.getDeploymentDescriptor();
   ...

   String newEARPath = null;
  }
}
```

- The following example provides a deployment step to wrap arbitrary content around an EAR file. Application management logic accepts only the EAR file for deployment. An extension is required if you want to input anything other than an EAR file to the deployment process.

```
package com.acme;

import com.ibm.websphere.management.deployment.core.DeploymentStep;
import com.ibm.websphere.management.deployment.core.DeployableObject;

public class EarWrapperStep extends DeploymentStep
{
  public void execute (DeployableObject dObject)
  {
   Archive archive = (Archive) dObject.getHandle();
   String newEARPath = null;
   // provide your logic to wrap the jar with the ear
   ...
```

```
      newEARPath = //;
      // Set the new ear path back into DeploymentContext
      this.getContext().getContextData()
        .put(DeploymentContext.RETURN_Object_key, newEARPath);


    }
  }
```

## Results

Through programming, you have plugged into the common deployment framework to extend application management operations.

## What to do next

You can extend other application management operations, or do any other administrative operations you choose.

# Chapter 7. Deploying and administering business-level applications

Deploying a business-level application consists of creating the business-level application on a Version 7.0 or later server.

## Before you begin

**Note:** A business-level application is an administration model that provides the entire definition of an application as it makes sense to the business. It is a WebSphere configuration artifact, similar to a server, that is stored in the product configuration repository. A business-level application can contain artifacts such as Java Platform, Enterprise Edition (Java EE) applications or modules, shared libraries, data files, and other business-level applications. You might use a business-level application to group related artifacts or to add capability to an existing application. For example, suppose you want to add capability provided in a Java archive (JAR) to a Java EE application already deployed on a product server. You can add that capability by creating a new business-level application and adding the JAR file and the deployed Java EE application to the business-level application. In some cases, you do not even need to change the deployed Java EE application configuration to add the capability.

Before creating a business-level application, you must develop the artifacts to go in the application and configure the target server. Before choosing a deployment target for the application, ensure that the target version is 7.0 or later.

## About this task

When creating a business-level application, you can configure the application enough to enable it to run on the server. Later, you can configure the application and its contents further, start or stop the application, and otherwise manage its activity.

The topics in this section describe how to deploy and administer a business-level application or its contents using the administrative console. You can also use programming or wsadmin scripting.

- Import assets to a repository.
- View, delete, update, or export assets.
- Create a business-level application.
- Start the application.
- Stop the application.
- Update the application and its configuration units.
- Delete the application.

## What to do next

After making changes to administrative configurations of your applications in the administrative console, ensure that you save the changes.

# Business-level applications

A business-level application is an administration model that provides the entire definition of an application as it makes sense to the business. A business-level application is a WebSphere configuration artifact, similar to a server or cluster, that is stored in the product configuration repository.

- Business-level application characteristics
- Comparisons to Java EE applications

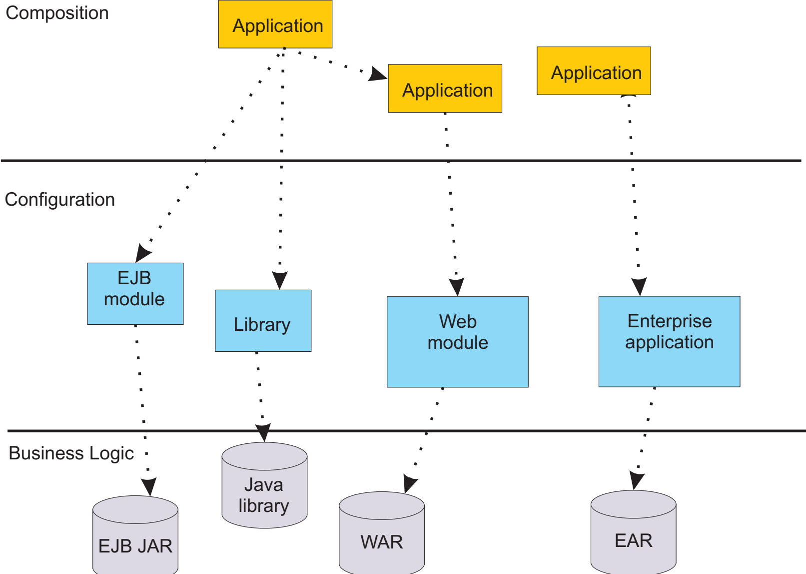
© IBM Corporation 2007 

## Business-level application characteristics

A business-level application has the following characteristics:

- A business-level application is an administration model of the definition of an enterprise-level application that consists of WebSphere and non-WebSphere artifacts. The business-level application might not explicitly manage the lifecycle of every artifact. It is a model for defining an application.
- A business-level application does not represent or contain application binary files. It is a configuration that lists one or more composition units, which represent the application binary files. A business-level application uses the binary files to run the application business logic. Administration of binary files is separate from administration of the application definition.
- A business-level application supports recursive composition by reference that facilitates hierarchical assembly of business-level applications and individual deployed artifacts within or outside a WebSphere product. The composition at its lowest level consists of configured instances of application binary files that run in a specific runtime environment such as an application server. Installable packages or archives, such as Java archives (JAR) or enterprise archive (EAR) files, typically deliver the business logic that these configured instances represent to corresponding runtime platforms.

The following diagram shows the composition model for business-level applications:




A business-level application does not introduce new programming, runtime, or packaging models:

- You do not need to change your application business logic. The business-level application function does not introduce new application programming interfaces (APIs).
- You do not need to change your application runtime settings. The product supports all of the runtime characteristics, such as security, class loading and isolation, required by individual programming models to which business components are written.
- You do not need to change your application packaging. There is no specific unique packaging model that provides a business-level application definition.

Typically, you first create an empty business-level application and then add composition units to it. The business-level application name must be unique within a cell. The business level application itself has minimal configuration data associated with it, solely the list of composition units, but individual composition units might save application-specific configuration data.

A business-level application is defined in the product configuration repository under *app_server_root*/cells/*cell_name*/blas/*business_level_application_name*/bver/BASE/bla.xml.

## Comparisons to Java EE applications

Business-level applications can consist of or aggregate Java Platform, Enterprise Edition (Java EE) applications and modules with non-Java EE artifacts. The contents of Java EE applications integrate with business-level application concepts for deployment and management of applications. Existing Java EE application management APIs continue to work after you add Java EE application or modules to a business-level application. The business-level application management API accepts Java EE contents and configurations and delegates to existing Java EE management APIs. Control operations such as starting and stopping a Java EE composition unit are delegated to ApplicationManager MBean on application servers that start and stop Java EE applications.

*Table 6. Java EE concepts compared to business-level application concepts*

| Java EE concept | Business-level application concept | Description |
|---|---|---|
| EAR or stand-alone module for deployment | Asset | Java EE application contents are assets. |
| Java EE application created at the end of application install | Composition unit | A Java EE application is in an enterprise archive (EAR) file. The product saves the EAR file in the product repository as a composition unit. |
| Java EE modules within the EAR file | Deployable units in the asset | Each module in the EAR file is a deployable unit that you can install on independent deployment targets. The EAR file is still managed as a single asset in its entirety. |
| Java EE application installation using the administrative console, programming, or wsadmin commands | Multiple business-level application management commands<br><br>During Java EE application deployment, you can specify the name of the business-level application to include the Java EE application. If the business-level application name is not set, the product creates a default business-level application with the same name as the Java EE application name. The product adds a composition unit with the same name as the Java EE application name under the business-level application. You can deploy multiple Java EE applications under a single business-level application. | You can make a Java EE application a business-level application and add it to another business-level application:<br>1. Install the Java EE application (EAR file) using the enterprise application installation console wizard, programming, or wsadmin. Keep the default selection to create a business-level application that has the same name as the Java EE application.<br>2. Create an empty business-level application.<br>3. Add the EAR file business-level application to the empty business-level application. The EAR file business-level application is a composition unit of the containing business-level application.<br><br>Or, you can make a Java EE application an asset and add it to another business-level application:<br>1. Import an EAR file as an asset. It has an asset type aspect of `Java EE ear`.<br>2. Create an empty business-level application.<br>3. Add the Java EE application asset to the business-level application. The EAR file asset is a composition unit of the containing business-level application.<br>4. Collect targets for each deployable unit (Java EE module). |
| Uninstall Java EE application | Multiple business-level application management commands | You delete the Java EE application composition unit from the business-level application:<br>1. Remove the composition unit for the Java EE application from the business-level application.<br>2. If the EAR file is an asset, delete the asset. |
| Start the Java EE application. | Start the composition unit. | Starting a business-level application starts any Java EE application in it. |

*Table 6. Java EE concepts compared to business-level application concepts (continued)*

| Java EE concept | Business-level application concept | Description |
| --- | --- | --- |
| Stop the Java EE application. | Stop the composition unit. | Stopping a business-level application stops any Java EE application in it. |

## Assets

An asset represents one or more application binary files that are stored in an asset repository. Typical assets include application business logic such as Java Platform, Enterprise Edition (Java EE) archives, library files, and other resource files.

An asset repository stores the binary files for the asset. The product configuration repository provides a default asset repository.

Assets in the configuration repository are managed by the product management domain. The configuration repository stores asset binary files in *app_server_root*/config/cells/*cell_name*/assets/*asset_name*/aver/*BASE*/bin/.

An asset name must be unique within a cell, the product administrative domain.

The product creates an asset.xml file when an asset is registered with the product configuration. The file contains information about the asset such as its name, destination location, and dependencies on other assets.

You must register files as assets before you can add them to one or more business-level applications. At the time of asset registration, you can import the physical application files into the product configuration repository or you can specify an external location where the asset resides.

## Composition units

A composition unit represents a configured asset in a business-level application. A composition unit enables the asset contents to interact with other assets in the application. It also enables the product run time to load and run asset contents.

The product supports three types of composition units:

**Asset composition units**
: Composition units created from assets by configuring each deployable unit of the asset to run on deployment targets.

**Shared library composition units**
: Composition units created from JAR-based assets by ignoring all the deployable objects from the asset and treating the asset JAR file as a library of classes.

**Business-level application composition units**
: Composition units created from business-level applications that are added to existing business-level applications.

A composition unit contains the following information:

- Configuration information that binds contents of an asset with a specific hosting run time and adds the configuration necessary for the run time to load and run the asset
- References to external services, components, or other resources that the asset uses
- Customized configurations for service definitions, references and other relevant configuration data

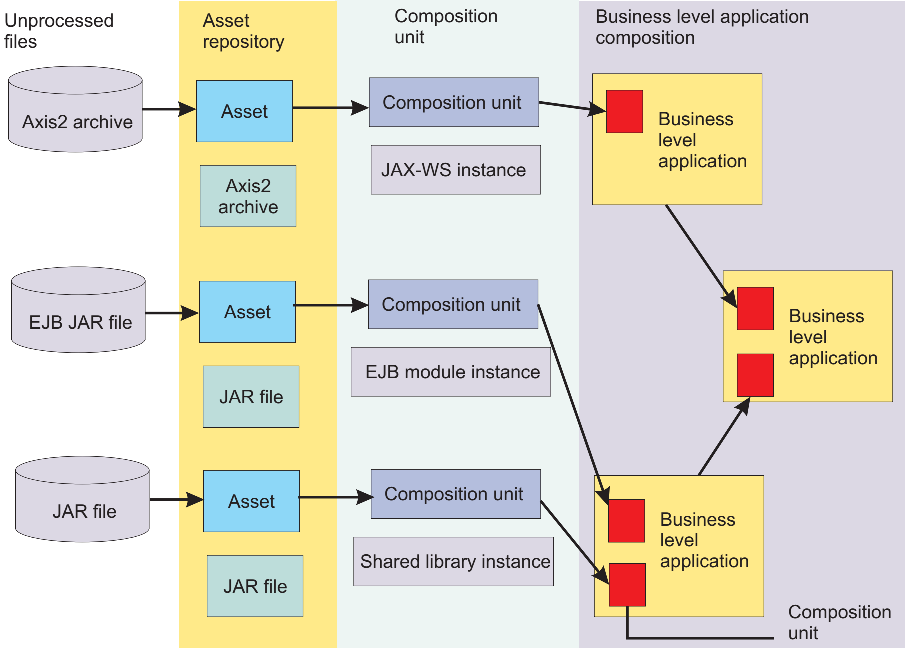
- A list of deployment targets or runtime environments along with the runtime environment-specific configuration where the composition unit runs.

For example, a composition unit for an enterprise bean (EJB) Java archive (JAR) asset is an EJB module instance that contains necessary EJB binding information, such as EJB Java Naming and Directory Interface (JNDI) names and `ejb-ref` resolutions, along with a list of application servers where the EJB JAR runs.

The product creates a composition unit from only one asset. However, multiple composition units can share a single asset. This is particularly useful in scenarios where different configurations use the same application binary files to provide different runtime behavior.

The following rules apply to a composition unit:
- A composition unit can exist only in a business-level application.
- Because a composition unit contains application-specific configuration and wiring information, multiple business-level applications cannot share an asset or shared library composition unit.

The following graphic shows the use of composition units in business-level applications. Assume that you have unprocessed files, such as archives, that you want to use in business-level applications. Before you can add the files to business-level applications, you must first import the files as assets, which adds the files to the product repository. Next, you add the assets to business-level applications, which creates composition units for the assets. Business-level applications can contain asset composition units, shared library composition units, or business-level composition units.




## Importing assets

You must register application business logic such as Java Platform, Enterprise Edition (Java EE) archives, libraries, and other resource files with the product configuration as assets before you can add the assets to one or more business-level applications. Importing an asset registers it with the product configuration.

### Before you begin

This topic assumes that you have one or more application binary files that you want to add to a business-level application. You must register those binary files as assets before you can add them to the business-level application.

## About this task

Before a business-level application that uses an asset can be started on the target run time, the asset binaries must be extracted to a deployer-defined location on the file system that is local to the target run time. Importing an asset extracts binaries to a location that is local to the target run time.

The application server run time that reads the asset binaries either at application start time or while serving an incoming client request determines the extraction format of the asset binaries. The extraction format might include unzipping of Java archive (JAR) or compressed (zip) files.

This topic describes how to import an asset using the administrative console. Alternatively, you can use the wsadmin tool or programming.

1. Click **Applications** → **New Application** → **New Asset** in the console navigation tree.
2. On the Upload asset page, specify the asset package to import.
   a. Specify the full path name of the asset.
   b. Click **Next**.
3. On the Select options for importing an asset panel, specify asset settings.

   You typically can click **Next** and use the default values.

   a. Optional: For **Asset description**, specify a brief description of the asset.
   b. Optional: For **Asset binaries destination URL**, specify the target location of the asset.

      This setting specifies the location to which the product extracts the asset. After an asset is imported, the product looks for the asset in this location when a running application uses the asset.

      If you do not specify a value, the product installs the asset to the default location, ${*profile_root*}/installedAssets/*asset_name*/BASE/.
   c. Optional: For **Asset type aspects**, examine the asset content type and version specified by the product. You cannot change this setting value.

      The type aspect typically denotes the type of application contents, such as a specification to which the application is written. For example, an enterprise bean (EJB) that supports the EJB Version 2.0 specification has the aspects `type=EJB,version=2.0`.

      If the type aspect is `none` and if the asset is a JAR file, then the product associates a `javarchive` type aspect with the asset by default.
   d. For **File permissions**, specify any file permissions that are set on asset binary files so the target run time can read or run the asset. Importing the asset extracts its binary files on the disk local to the target runtime environment.

      Try importing the asset using the default value. For detailed information on the **File permissions** setting, refer to the Select options for importing an asset panel online help.
   e. For **Current asset relationships**, add assets that the asset you are importing needs to run or remove assets that are not needed.

      When the product imports a JAR asset, the product detects asset relationships automatically by matching the dependencies defined in the JAR manifest with the assets that are already imported into the administrative domain.
   f. For **Validate asset**, specify whether the product validates the asset.

      The setting is deselected by default. This **false** (**no**) value is appropriate for most assets. Only select **true** (**yes**) to validate an asset when needed.

      The product does not save the value specified for **Validate asset**. Thus, if you select to validate the asset (**yes**) now and later update the asset, when you update the asset you must enable this setting again for the product to validate the updated files.
   g. Click **Next**.
4. On the Summary page, click **Finish**.

### Results

Several messages are displayed, indicating whether your asset is imported successfully.

An asset can contain multiple deployable objects as defined by the application contents of that asset. A *deployable object* is a part of the asset that you can map to a deployment target such as an application server. If the product imports the asset successfully, then appropriate deployable objects are identified in the asset and are further used when a composition unit is created from that asset.

If the asset importing is not successful, read the messages and try importing the asset again. Correct the values noted in the messages.

### What to do next

If the product imports the asset successfully and displays the list of assets on the Assets page, then click **Save**.

Add a composition unit to a business-level application using the asset that you imported. An asset included in a business-level application is represented by a composition unit.

## Upload asset settings

Use this panel to specify the asset to register with the asset repository. You can add registered assets to a business-level application.

To view this administrative console panel, click **Applications** → **New application** → **New Asset**.

Importing an asset registers the asset with the asset repository.

The product manages the contents of a registered asset as a single entity. The contents of a registered asset must be accessible to application servers, Web servers and other runtime environments that use the asset.

During asset importing, asset files typically are uploaded from a client workstation running the browser to the server running the administrative console, where they are registered. In such cases, use the Web browser running the administrative console to select files to upload to the server.

### Path to the asset

Specifies the fully qualified path to the asset.

Specify one of the following supported assets:
- A single file, such as an enterprise bean (EJB) file
- An archive of files, such as a Java archive (JAR) or a compressed (zip) file
- An archive of archives, such as an enterprise archive (EAR) or shared library file

Use **Local file system** if the browser and asset files are on the same machine (whether or not the server is on that machine, too).

Use **Remote file system** if the asset file resides on any node in the current cell context. Only supported assets are shown during the browsing. Also use **Remote file system** to specify an asset file that is already residing on the machine running the application server. For example, the field value might be *profile_root*/installableApps/my_bean.ejb. After the asset file is transferred, the **Remote file system** value shows the path of the temporary location on the server.

# Asset settings

Use this page to specify options for the registration of an asset with the asset repository. Default values for the options are used if you do not specify a value.

To view this administrative console page, click **Applications** → **Application Types** → **Assets** → ***asset_name***. This page is similar to the Select options for importing an asset panel on the asset import and update wizards.

## Asset name

Specifies a logical name for the asset. An asset name must be unique within a cell and cannot contain an unsupported character.

An asset name cannot begin with a period (.), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

<table>
<tr><th colspan="3">Unsupported characters</th></tr>
<tr><td>/ forward slash</td><td>$ dollar sign</td><td>’ single quote mark</td></tr>
<tr><td>\ backslash</td><td>= equal sign</td><td>″ double quote mark</td></tr>
<tr><td>* asterisk</td><td>% percent sign</td><td>| vertical bar</td></tr>
<tr><td>, comma</td><td>+ plus sign</td><td>< left angle bracket</td></tr>
<tr><td>: colon</td><td>@ at sign</td><td>> right angle bracket</td></tr>
<tr><td>; semi-colon</td><td># hash mark</td><td>& ampersand (and sign)</td></tr>
<tr><td>? question mark</td><td colspan="2">]]> No specific name exists for this character combination</td></tr>
</table>

This **Asset name** field is the same as the **Name** setting on an Assets page.

**Data type** String

## Asset description

Specifies a description for the asset.

## Asset binaries destination URL

Specifies the directory to which the product imports the asset file.

**Data type** String
**Units** Full path name

## Asset type aspects

Specifies the type of asset content. Examples of asset type include Java archive (JAR) files, shared libraries, and enterprise application archive (EAR) files.

The asset type suggests the content of the asset. For example, an asset packaged as a JAR file might contain a Web module, portlet and Web service.

This setting is read-only. You cannot edit this setting.

**Data type** String
**Units** File type
**Default** none

## File permissions

Specifies access permissions for asset binaries that the product expands to the asset binaries destination URL.

You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the list. List selections overwrite file permissions set in the text field.

You can set one or more of the following file permission strings in the list. Selecting multiple options combines the file permission strings.

| List option | File permission string set |
|---|---|
| **Allow all files to be read but not written to** | `.*=755` |
| **Allow executables to execute** | `.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755` |
| **Allow HTML and image files to be read by everyone** | `.*\.htm=755#.*\.html=755#.*\.gif=755#.*\.jpg=755` |

Instead of using the list to specify file permissions, you can specify a file permission string in the text field. File permissions use a string that has the following format:

*file_name_pattern=permission#file_name_pattern=permission*

where *file_name_pattern* is a regular expression file name filter (for example, `.*\\.jsp` for all JSP files), *permission* provides the file access control lists (ACLs), and `#` is the separator between multiple entries of *file_name_pattern* and *permission*. If `#` is a character in a *file_name_pattern* string, use `\#` instead.

If multiple file name patterns and file permissions in the string match a uniform resource identifier (URI) within the asset, then the product uses the most stringent applicable file permission for the file. For example, if the file permission string is `.*\\.jsp=775#a.*\\.jsp=754`, then the abc.jsp file has file permission 754.

**Note:** Using regular expressions for file matching pattern compares an entire string URI against the specified file permission pattern. You must provide more precise matching patterns using regular expressions as defined by Java programming API. For example, suppose the product processes the following directory and file URIs during a file permission operation:

| | |
|---|---|
| 1 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war` |
| 2 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp` |
| 3 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF/MANIFEST.MF` |
| 4 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/WEB-INF/classes/MyClass.class` |
| 5 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/mydir/MyClass2.class` |
| 6 | `/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF` |

The file pattern matching results are:
- `MyWarModule.war` does not match any of the URIs
- `.*MyWarModule.war.*` matches all URIs
- `.*MyWarModule.war$` matches only URI 1
- `.*\\.jsp=755` matches only URI 2
- `.*META-INF.*` matches URIs 3 and 6
- `.*MyWarModule.war/.*/.*\.class` matches URIs 4 and 5

If you specify a directory name pattern for **File permissions**, then the directory permission is set based on the value specified. Otherwise, the **File permissions** value set on the directory is the same as its parent. For example, suppose you have the following file and directory structure:

```
/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
```

and you specify the following file pattern string:

```
.*MyApp.ear$=755#.*\.jsp=644
```

The file pattern matching results are:
- Directory MyApp.ear is set to 755
- Directory MyWarModule.war is set to 755
- Directory MyWarModule.war is set to 755

**Note:** Regardless of the operation system, always use a forward slash (/) as a file path separator in file patterns.

Access permissions specified here are at the asset level. You can also specify access permissions for asset binaries in the node-level configuration. The node-level file permissions specify the maximum (most lenient) permissions that can be given to asset binaries. Access permissions specified here at asset level can only be the same as or more restrictive than those specified at the node level.

| | |
|---|---|
| **Data type** | String |

## Current asset relationships

Specifies the assets to which this asset is related.

To add or remove a relationship, use the Manage relationships panel:
1. Click **Manage Relationships** to access the Manage relationships panel. The **Selected** list on the right lists the current asset relationships.
2. To add a relationship, select an asset in the **Available** list on the left and click **>>**.
3. To remove a relationship, select an asset in the **Selected** list on the right and click **<<**.
4. Click **OK**.

| | |
|---|---|
| **Data type** | String |
| **Default** | none |

## Validate asset

Specifies whether the product examines the asset references specified during asset importing or updating and, if validation is enabled, warns you of incorrect references or fails the operation.

An asset typically refers to resources using data sources for container-managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the asset is defined in the scope of the deployment target of that asset.

Select `true` (enable the check box) for resource validation and to stop operations that fail as a result of incorrect resource references. Select `false` (empty check box) for no resource validation.

| | |
|---|---|
| **Data type** | String |
| **Default** | false (empty check box) |

# Managing assets

After application binary files are imported and registered with the product management domain as assets, you can view, update and export those assets.

## Before you begin

Import one or more assets. The name of each imported assets is shown on the list of assets on the administrative console Assets page.

## About this task

You can view the contents of assets, update assets, remove assets from the product management domain, or export copies of assets to a target location. This topic describes how to perform these asset management operations from the administrative console Assets page. Alternatively, you can use programming or the wsadmin tool.

- View or edit asset settings.
  1. Go to the administrative console Assets page.

     Click **Applications** → **Application Types** → **Assets**.
  2. Click the asset name in the list of assets. The Asset settings page displays the values that are specified for the asset.
  3. Optional: Change the asset settings as needed and click **OK** to save the changes.
- Remove one or more assets from the product management domain.
- Update the contents of an asset.
- Export an asset to a target location.

## What to do next

Create a business level application and add the asset to the business-level application.

# Asset collection

Use this page to view a list of assets in the asset repository and to manage those assets. After importing an asset, you can add the asset to a business-level application.

Assets include Java archive (JAR) and compressed files that are used by applications installed on a server.

To view this administrative console page, click **Applications** → **Application Types** → **Assets**.

To view the values specified for an asset, click the asset name in the list. The displayed asset settings page shows the values specified. On the settings page, you can change existing asset values.

To manage an asset, enable the **Select** check box beside the asset name in the list and click a button:

| Button | Resulting action |
|---|---|
| **Import** | Opens a wizard that helps you add an asset to the asset repository. |
| **Delete** | Removes the asset from the asset repository and deletes the asset binaries from the file system of all nodes where the assets are installed.<br><br>On single-server installations, deletion occurs after the configuration is saved. |

| Button | Resulting action |
| --- | --- |
| **Update** | Opens a wizard that helps you update asset files. You can replace a file or module that exists on the server with a file or module that has the same name. Or you can add a new file or module, provided the new file or module does not have the same name as an asset that already exists on the server. |
| **Export** | Accesses the Export asset page, which you use to export an asset to a file at a location of your choice. Use the **Export** action to back up an asset. |

### Name

Specifies the name of the asset. Asset names must be unique within a cell and cannot contain an unsupported character.

### Description

Specifies a description for the asset.

## Updating assets

You can update application binary files that are assets registered with the product management domain.

### Before you begin

Import one or more assets. The file name of each deployable object in the imported assets is shown on the list of assets on the administrative console Assets page.

### About this task

You can update all of part of the contents of assets that are in the product management domain. This topic describes how to update an asset using the administrative console Update asset wizard. Alternatively, you can update assets using programming or the wsadmin tool.

1. Go to the Update asset wizard.
   a. Click **Applications** → **Application Types** → **Assets** to access the Assets page.
   b. Select the check box beside the asset that you want to update.
   c. Click **Update**.
2. On the Update asset panel, specify whether you want replace an entire asset or update its contents and, as needed, the replacement file or module.
   a. Select an update option.

      You can update asset contents by adding, deleting, or updating a single file or module in the asset, or by merging multiple files or modules. Update options include the following:
      - Replace entire asset
      - Replace specific asset contents
      - Add module or file to asset
      - Remove file or module from asset
      - Merge asset contents

      The online help for the Update asset panel describes the options.
   b. If you are updating specific asset contents or removing a file or module, specify the path beginning with the asset archive file.

      For **Specify the path beginning with the asset archive file**, specify a relative path to the file that starts from the root of the asset file. For example, if the file is located at com/company/greeting.class in module hello.jar, specify a relative path of hello.jar/com/company/greeting.class.
   c. If you are updating the entire asset, updating an asset file or module, or merging asset contents, specify the full path name of the new file or module.
   d. Click **Next**.

3. On the Select options for updating an asset page, specify asset settings and click **Next**.

   The online help for the Select options for importing an asset panel describes the settings.
4. On the Summary page, click **Finish**.

## Results

If you update an asset packaged as a library JAR file that is not a Java Platform, Enterprise Edition (Java EE) archive, then the product automatically distributes the updated asset to all of the composition units that use the asset.

However, if you update a Java EE asset, then the product does not automatically distribute the updated Java EE archive to composition units created from that asset, which are Java EE applications. You must select every Java EE application created from that asset and use the **Update** button to update the Java EE application individually by specifying the update contents.

## What to do next

Create a business-level application and add the asset to the business-level application.

## Update asset settings

Use this panel to select whether you want replace an entire asset or update its contents. You can update asset contents by adding, deleting, or updating a single file or module in the asset, or by merging multiple files or modules into an asset. Updating an asset registers the updated files with the product management domain.

To view this administrative console panel, click **Applications** → **Application Types** → **Assets**, select the asset to update, and then click **Update**.

The product manages the contents of a registered asset as a single entity. The contents of a registered asset must be accessible to application servers, Web servers and other runtime environments that use the asset.

When you replace an asset or update an asset by adding a file or module, asset files typically are uploaded from a client workstation running the browser to the server machine running the administrative console, where they are registered. In such cases, use the Web browser running the administrative console to select files to upload to the server machine.

The specified asset that you are installing must be one of the following supported assets:
- A single file, such as an enterprise bean (EJB) file
- An archive of files, such as a Java archive (JAR) or a compressed (zip) file
- An archive of archives, such as an enterprise archive (EAR) or shared library file

***Replace entire asset:***

Under **Select the type of update to perform**, specifies to replace the entire asset installed on the server with a new (updated) asset.

After selecting this option, specify whether the asset is on a local or remote file system and the full path name of the asset. The path provides the location of the updated asset before installation.

Use **Local file system** if the browser and asset files are on the same machine (whether or not the server is on that machine, too).

Use **Remote file system** if the asset file resides on any node in the current cell context. Only supported assets are shown during the browsing. Also use **Remote file system** to specify an asset file that is already residing on the machine running the application server. For example, the field value might be

*profile_root*/installableApps/my_bean.ejb. After the asset file is transferred, the **Remote file system** value shows the path of the temporary location on the server.

***Replace specific asset contents:***

Under **Select the type of update to perform**, specifies to replace a file or module of the asset installed on the server.

After selecting this option, do the following:
1. For **Specify the path beginning with the asset archive file**, specify a relative path to the file that starts from the root of the asset file. For example, if the file is located at com/company/greeting.class in module hello.jar, specify a relative path of hello.jar/com/company/greeting.class.
2. Specify whether the asset is on a local or remote file system and the full path name of the asset. The path provides the location of the updated asset before installation.
3. Click **Next**.

The **Replace entire asset** description describes options for specifying the full path name of an asset or file to add using **Local file system** and **Remote file system** options.

***Add a module or file to an asset:***

Under **Select the type of update to perform**, specifies to add a file to the asset installed on the server.

After selecting this option, do the following:
1. For **Specify the path beginning with the asset archive file**, specify a relative path to the file that starts from the root of the asset file. For example, if the file is located at com/company/greeting.class in module hello.jar, specify a relative path of hello.jar/com/company/greeting.class.
2. Specify whether the asset is on a local or remote file system and the full path name of the asset. The path provides the location of the updated asset before installation.

The **Replace entire asset** description describes options for specifying the full path name of an asset or file to add using **Local file system** and **Remote file system** options.

***Remove a file or module from an asset:***

Under **Select the type of update to perform**, specifies to remove a file or module from the asset installed on the server.

After selecting this option, do the following:
1. For **Specify the path beginning with the asset archive file**, specify a relative path to the file to be removed that starts from the root of the asset file. For example, if the file is located at com/company/greeting.class in module hello.jar, specify a relative path of hello.jar/com/company/greeting.class.
2. Click **Next**.

***Merge asset contents:***

Under **Select the type of update to perform**, specifies to compare the new file or module with the file or module of the asset installed on the server. If the file or module exists, it is replaced. Otherwise, it is added to the installed asset.

After selecting this option, specify whether the new file or module is on a local or remote file system and the full path name of the file or module. The path provides the location of the updated asset before installation.

The **Replace entire asset** description describes options for specifying the full path name of a file or module to merge using **Local file system** and **Remote file system** options.

## Deleting assets

You can remove application binary files that are registered as assets from the product management domain.

### Before you begin

Import one or more assets. The name of each imported asset is shown on the list of assets on the administrative console Assets page.

### About this task

You can remove assets from the product management domain, provided the asset does not have an existing composition unit. If an asset has one or more composition units defined in the management domain, then you cannot delete that asset until those composition units are removed.

This topic describes how to delete assets using the administrative console. Alternatively, you can use programming or the wsadmin tool.

1. Go to the Delete asset page.
   a. Click **Applications** → **Application Types** → **Assets** to access the Assets page.
   b. Select the check box beside the asset that you want to delete.
   c. Click **Delete**.
2. On the Delete asset page, click **OK** to confirm that you want the specified asset removed from the product management domain.

   Click **Cancel** to return to the Assets page and not delete the asset.

### Results

The product deletes the asset from the product management domain.

### What to do next

On the Assets page, verify that the deleted asset is no longer in the list of imported assets.

## Exporting assets

After application binary files are imported and registered with the product management domain as assets, you can export those assets.

### Before you begin

Import one or more assets. The file name of each deployable object in the imported assets is shown on the list of assets on the administrative console Assets page.

### About this task

You can export copies of assets to a target location. Exporting stores application binary files, enabling you to back up the files or edit them. The file resulting from exporting an asset contains configuration information for the asset.

This topic describes how to export an asset from the administrative console Assets page. Alternatively, you can use programming or the wsadmin tool.

1. Go to the Export asset page.
   a. Click **Applications** → **Application Types** → **Assets** to access the Assets page.
   b. Select the check box beside the asset that you want to export.
   c. Click **Export**.
2. On the Export asset page, click the asset name or identifier.
   To cancel the export operation and return to the Assets page, click **Back**.
3. Specify the target location for the asset file.

### What to do next

Examine the target file to verify that the asset exported correctly. You can later edit this file and import the edited asset.

## Creating business-level applications

You can create an empty business-level application and then add assets, shared libraries, business-level applications, and other artifacts as composition units to the empty business-level application.

### Before you begin

Configure each target application server as needed. You must deploy a business-level application to a Version 7.0 server.

Optionally, determine what assets or other files that you want to add to your business-level application and whether your application files can run on your deployment targets.

### About this task

You can create business-level applications using the administrative console, programming, or the wsadmin tool.

1. Select a way to create your business level application.

*Table 7. Ways to create business level applications*

| Option | Method |
|---|---|
| Administrative console business-level application creation wizard<br><br>See "Creating business-level applications with the console" on page 173. | Click **Applications** → **New application** → **New Business-level Application** and follow instructions in the wizard. |
| Administrative console Java Platform, Enterprise Edition (Java EE) application installation wizard<br><br>See "Installing enterprise application files with the console" on page 36. | Click **Applications** → **New application** → **New Enterprise Application** and follow instructions in the wizard.<br><br>The product creates a new business-level application with the enterprise application that you install or makes the enterprise application a composition unit of an existing business-level application. See the **Business-level application name** setting on the Select installation options wizard panel. |

2. Create your business-level application using the administrative console, programming or wsadmin.
3. Save the changes to your administrative configuration.

### Results

The name of the application is shown in the list on the Business-level applications page.

### What to do next

After you create a business-level application, you can do the following to add composition units to it:
1. Import any assets needed by your business-level application.
2. Add assets, shared libraries, or other business-level applications as composition units.
3. Save the changes to your administrative configuration.
4. Start the business-level application.

If the application does not run as desired, edit the application configuration, then save and run it again.

## Creating business-level applications with the console

You can create an empty business-level application and then add assets or business-level applications as composition units to the empty business-level application.

### Before you begin

Before you create a business-level application, decide upon an application name. Optionally, determine which assets, shared libraries, or business-level applications that the new business-level application needs.

### About this task

This topic describes how to create an empty business-level application and then add assets as composition units to the application using the administrative console. Alternatively, you can use programming or the wsadmin tool.

You can add an asset or shared library composition unit to multiple business-level applications. However, each composition unit for the same asset must have a unique composition unit name. You can add a business-level application composition unit to more than one business-level application.

1. Create an empty business-level application.
   a. Click **Applications** → **New application** → **New Business Level Application**.
   b. On the New business-level application panel, specify a unique name for the application and a description, and then click **OK**.
   c. On the business-level application settings page, click **Save**.

   The name and description are shown in the list of applications on the Business-level applications page. Because the application is empty, its status is `Unavailable`.
2. Optional: Add one or more assets, non-Java EE shared libraries, or business-level applications to a business-level application. The product adds these assets as composition units of your business-level application.

   If the asset that you want to add to your business-level application is a Java Platform, Enterprise Edition (Java EE) application or module that is not yet deployed, see step 3. If the asset is a Java EE shared library, see step 4.
   a. Import the assets or create the business-level applications that you want to add to the business-level application.
   b. Go to the business-level application settings page.

      Click **Applications** → **Application Types** → **Business-level applications** → ***application_name***.
   c. On the business-level application settings page, specify the type of composition unit to add.

- To add an asset, under **Deployed assets**, click **Add** → **Add Asset**.
- To add a shared library, under **Deployed assets**, click **Add** → **Add Shared Library**.
- To add a business-level application, under **Business-level applications**, click **Add**.

d. On the Add panel, select a unit from the list of available units, and then click **Continue**.

If you are adding one or more deployable unit assets and you have multiple imported assets available, you can select more than one deployable unit.

e. On the Set options panel, change the composition unit settings as needed, and then click **Next**.

This panel is not shown when you add a Java EE asset as a shared library or if you have multiple deployable unit assets. If the application installation or update wizard displays and you want to add a Java EE asset as a shared library, see step 4.

f. On the Map composition unit to a target panel, change the deployment target as needed, and then click **Next**.

This panel is not shown when you add a business-level application.

g. If you are adding one or more deployable unit assets, specify composition unit relationship options.

See “Deployable unit relationship settings” on page 181.

h. On the Summary page, click **Finish**. Several messages are displayed, indicating whether the product adds the unit to the business-level application successfully. A message having the format `Completed res=[WebSphere:cuname=`*`unit_name`*`,cuedition=`*`version`*`]` indicates that the addition is successful. Click **Manage application**.

If the product adds the unit successfully, the name of the unit is shown on the list of composition units on the Adding composition unit to the business-level application page.

If the unit addition is not successful, read the messages and try adding the unit again. Correct the problems noted in the messages.

i. On the Adding composition unit to the business-level application page, click **Save**.

The product creates composition units for the asset, shared library, or business-level application. The unit names are shown in lists of composition units on the settings page of your business-level application. To view the settings page, click **Applications** → **Application Types** → **Business-level applications** → ***your_application_name***.

3. Optional: Install a Java EE application or module, and add it as a composition unit to your business-level application.

When installing an enterprise archive (EAR) file or a stand-alone Java EE module using the application installation wizard, you can specify a business-level application to which to add the EAR file or module. You can also specify relationships to any shared libraries that your Java EE application or module uses. The product creates composition units that represent those relationships.

a. Click **Applications** → **New application** → **New Enterprise Application**.

b. On the first Preparing for the application installation panel, specify the Java EE application or module to install and click **Next**.

c. On the second Preparing for the application installation panel, select **Detailed - Show all installation options and parameters**, specify whether to generate default bindings and mappings as needed for the application or module, and click **Next**.

d. On the Select installation options panel of the wizard, select your business-level application for **Business-level application name** and click **Next**. The product creates a composition unit that has the same name as the Java EE application or module and adds the unit to your business-level application.

If you do not specify a value for **Business-level application name**, then the product creates a default business-level application that has the same name as the Java EE application that you are installing. The product does not add the Java EE application as a composition unit to the business-level application that you created in step 1.

e. Optional: On the Map shared library relationship panel of the wizard, specify relationship identifiers and composition unit names for shared libraries that modules in your Java EE application use. The product creates a composition unit for each shared library relationship in your business-level application.

   You can map shared library relationships when installing your Java EE application or module or, after installation, return to the Map shared library relationship panel and specify shared library relationships. See step 4.

f. Complete the other application installation wizard options as needed to install the Java EE application or module.

The product creates composition units for the application, module, or shared library relationships. The unit names are shown in lists of composition units on the settings page of your business-level application. To view the settings page, click **Applications** → **Application Types** → **Business-level applications** → ***your_application_name***.

4. Optional: After installation of a Java EE application or module, specify composition units for relationships to shared libraries used by your business-level application on the Map shared library relationship panel of the application installation or update wizard.
   a. If you have not done so already, import a Java EE asset such as an enterprise bean (EJB) or Web module (WAR) that uses a shared library file.

      If the product displays `javaarchive` for **Asset type aspects** on the asset settings page, continue to step 4b.

      If the product does not display `javaarchive` for **Asset type aspects** on the asset settings page, then the asset is not a Java EE asset. Use step 2 to add a shared library to your business-level application.
   b. Go to a settings page for your business-level application.

      Click **Applications** → **Application Types** → **Business-level applications** → ***your_application_name***.
   c. Under **Deployed assets**, click **Add** → **Add Shared Library**.
   d. On the Add composition unit panel, select the Java EE asset that you imported and then click **Continue**.

      The Java EE application installation or update wizard displays. Select the Java EE application or module that uses the asset, and complete the steps in the wizard.
   e. On the Select installation options panel of the wizard, select your business-level application for **Business-level application name**.
   f. On the Map shared library relationship panel of the wizard, specify a relationship identifier and composition unit name for the asset.
   g. Complete the other wizard options as needed.

   The product creates a composition unit for the shared library relationship. The unit name is shown in the list of deployed asset composition units on the settings page of your business-level application.

## Results

The name of your business-level application is shown on the Business-level applications page in the list of applications.

## What to do next

After you create the application, save the changes to your configuration and start the application as needed.

## Business-level application collection

Use this page to view and manage business-level applications.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications**.

To view the values specified for an application configuration, click the application name in the list. The displayed application settings page shows the values specified. On the settings page, you can change existing configuration values and link to additional console pages that assist you in configuring the application.

To manage a business-level application, enable the **Select** check box beside the application name in the list and click a button:

| Button | Resulting action |
| --- | --- |
| **Start** | Attempts to run the application. After the application starts successfully, the state of the application changes to *Started* if the application starts on all deployment targets, else the state changes to *Partial Start*. |
| **Stop** | Attempts to stop the processing of the application. After the application stops successfully, the state of the application changes to *Stopped* if the application stops on all deployment targets, else the state changes to *Partial Stop*. |
| **New** | Opens a wizard that helps you add assets, shared libraries, or business-level applications as composition units to your application. |
| **Delete** | Deletes the application from the product configuration repository and deletes the application binaries from the file system of all nodes where the application modules are installed.<br><br>On single-server installations, deletion occurs after the configuration is saved. |

***Name:***

Specifies the name of the business-level application. Application names must be unique within a cell and cannot contain an unsupported character.

***Description:***

Specifies a description for the business-level application.

***Status:***

Indicates whether the application deployed on the application server is started, stopped, or unknown.

| | Status | Description |
| --- | --- | --- |
| | **Started** | Application is running. |
| | **Partial start** | Application is in the process of changing from a *Stopped* state to a *Started* state. Application is starting to run but is not fully running yet. Or, it cannot fully start because a server mapped to one or more application modules is stopped. |
| | **Stopped** | Application is not running. |
| | **Partial stop** | Application is in the process of changing from a *Started* state to a *Stopped* state. Application has not stopped running yet. |
| | **Unknown** | Status cannot be determined. |
| | **Pending** | Status is temporarily unknown pending an event that a user did not initiate, such as pending an asynchronous call. |
| | **Not applicable** | Application does not provide information as to whether it is running. |

The status of an application on a Web server is always **Unknown**.

## New business-level application settings

Use this panel to name and describe a new business-level application.

To view this administrative console panel, click **Applications** → **New application** → **New Business-level Application**.

***Name:***

Specifies a logical name for the business-level application. An application name must be unique within a cell and cannot contain an unsupported character.

An application name cannot begin with a period (.), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

| **Unsupported characters** | | |
|---|---|---|
| **/** forward slash | **$** dollar sign | **’** single quote mark |
| **** backslash | **=** equal sign | **″** double quote mark |
| ***** asterisk | **%** percent sign | **\|** vertical bar |
| **,** comma | **+** plus sign | **<** left angle bracket |
| **:** colon | **@** at sign | **>** right angle bracket |
| **;** semi-colon | **#** hash mark | **&** ampersand (and sign) |
| **?** question mark | **]]>** No specific name exists for this character combination | |

**Data type** String

***Description:***

Specifies a description for the application.

This field is the same as the **Description** setting on a Business-level applications page.

## Shared library relationship and mapping settings

Use the Shared library relationship and Shared library relationship mapping pages to specify relationship identifiers and composition unit names for shared libraries that modules in your enterprise application reference. When installing your enterprise application, the product creates a composition unit for each shared library relationship in the business-level application that you specified on the Select installation options panel of the application installation wizard.

To view this console panel in a wizard, click **Applications** → **Install new application** → **New Enterprise Application** → ***application_path*** → **Next** → **Detailed - Show all installation options and parameters** → **Next** → ***application_name*** → **Step: Map shared library relationships**.

After installation, click **Applications** → **Application Types** → **WebSphere enterprise applications** → **Shared library relationships**

To map library files used in a business-level application with an application or Web module, use the Shared library relationship mapping page:

1. Click **Reference shared libraries**.
2. Note the application or module in **Map libraries to the application or module listed**. You are associating library files with that application or module.
3. From the **Available** list, select one or more libraries that the application or module uses.

4. Click **>>** to add them to the **Selected** list.
5. To remove an association, select one or more libraries in the **Selected** list and click **<<**.
6. Click **OK**.

***Module:***

Specifies the name of the module associated with the shared libraries.

***URI:***

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

***Relationship identifers:***

Specifies an identifier for a module shared library relationship. The product assigns an identifier to the composition unit that it creates for the shared library relationship in the business-level application.

***Composition unit names:***

Specifies a composition unit name for the shared library relationship. The product uses this value to name the composition unit that it creates for the shared library relationship in the business-level application that you specified on the Select installation options panel of this wizard.

This setting is only in the application installation and update wizards.

***Match target:***

Specifies whether the product maps the composition unit for the shared library relationship to the same deployment target as the business-level application.

## Add composition unit settings

Use this panel to specify options for the composition unit to be added to the business-level application. The product assigns a default value for an option when you do not specify a value.

To view this administrative console panel, click **Applications** → **Application Types** → **Business-level applications** → ***business-level_application_name*** → **Add** → **Add *unit_type***.

***Name:***

Specifies the name of the composition unit to be added to the business-level application.

The table lists available composition units. Select a unit from this list.

***Description:***

Specifies a description for the composition unit.

## Add asset settings

Use this panel to add one or more assets to a business-level application.

To view this administrative console panel, click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → **Add** → **Add Asset**.

***Deployable units:***

Specifies the imported assets available for use in a business-level application. The list of deployable units includes only imported assets, and not shared libraries or business-level applications.

From this list, select one or more deployable units to add as composition units to your business-level application.

## Set options settings

Use this panel to specify options for the composition unit to be added to the business-level application. The product supplies default values for the options if you do not specify a value.

To view this administrative console panel, click **Applications** → **Application Types** → **Business-level applications** → ***application_name***. On the business-level application settings page, specify the type of composition unit to add:

- To add an asset, under **Deployed assets**, click **Add** → **Add Asset**.
- To add a shared library, under **Deployed assets**, click **Add** → **Add Shared Library**.
- To add a business-level application, under **Business-level applications**, click **Add**.

***Backing identifier:***

Specifies a unique identifier for a composition unit that is registered in the application management domain.

The identifier has the format: `WebSphere:`*`unit_type`*`name=`*`unit_name`*`,`*`unit_type`*`version=`*`version_number`*. For example, for the MyApp.jar asset, the backing identifier might be `WebSphere:assetname=MyApp.jar`.

| | |
|---|---|
| **Data type** | String |
| **Units** | Composition unit identifier |

***Name:***

Specifies the name of the composition unit.

For example, for the MyApp.jar asset, the name might be MyApp.jar.

A unit name cannot begin with a period (`.`), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

<table>
<tr><th colspan="3">Unsupported characters</th></tr>
<tr><td>/ forward slash</td><td>$ dollar sign</td><td>' single quote mark</td></tr>
<tr><td>\ backslash</td><td>= equal sign</td><td>" double quote mark</td></tr>
<tr><td>* asterisk</td><td>% percent sign</td><td>| vertical bar</td></tr>
<tr><td>, comma</td><td>+ plus sign</td><td>< left angle bracket</td></tr>
<tr><td>: colon</td><td>@ at sign</td><td>> right angle bracket</td></tr>
<tr><td>; semi-colon</td><td># hash mark</td><td>& ampersand (and sign)</td></tr>
<tr><td>? question mark</td><td colspan="2">]]> No specific name exists for this character combination</td></tr>
</table>

| | |
|---|---|
| **Data type** | String |

***Description:***

Specifies a description for the composition unit.

*Starting weight:*

Specifies the order in which composition units are started when the server starts. The starting weight is like the startup order. The composition unit with the lowest starting weight is started first.

The value that you set for **Starting weight** determines the importance or weight of a composition unit within the business-level application. For example, for the most important composition unit within a business-level application, specify 1 for **Starting weight**. For the next most important composition unit within the business-level application, specify 2 for **Starting weight**, and so on.

| | |
|---|---|
| **Data type** | Integer |
| **Default** | 1 |
| **Range** | 0 to 2147483647 |

*Start composition unit upon distribution:*

Specifies whether to start the composition unit after the product distributes the composition unit to other locations.

The default is not to start the composition unit.

| | |
|---|---|
| **Data type** | Boolean |
| **Default** | false |

*Restart behavior on update:*

Specifies whether the product restarts deployment targets after updates to the composition unit.

Usually, a composition unit is mapped to one or more deployment targets. This setting determines whether the product restarts those targets after editing the composition unit.

| Option | Description |
|---|---|
| ALL | The product restarts each target node of the composition unit after editing the composition unit. |
| DEFAULT | The product restarts the nodes controlled by the sync plug-ins after editing the composition unit. |
| NONE | The product does not restart nodes after editing the composition unit. |

## Map target settings

Use this panel to map a composition unit to a deployment target. The product assigns a default target when you do not specify a target.

To view this administrative console panel, click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → ***composition_unit_name*** → **Modify Target**. The Map target page is similar to the Map composition unit to a target panel in the add composition unit wizard.

On single-server products, a deployment target can be an application server or Web server.

On this panel, map a composition unit to one or more desired targets.

*Current targets:*

Specifies the existing deployment targets for the composition unit.

*Available:*

Lists the names of available deployment targets. This list is the same for every composition unit that is registered in the cell.

From this list, select only appropriate deployment targets for a composition unit.

If the unit calls a Version 7.x application programming interface (API) or uses a 7.x feature, then you must map the unit to a 7.x deployment target. If the unit supports Java Platform, Enterprise Edition (Java EE) 5, then you must map the unit to a 7.x deployment target. If the unit supports Java 2 Platform, Enterprise Edition (J2EE) 1.4, then you must map the unit to a 6.x or 7.x deployment target. You can map units that call a 6.x API or use a 6.x feature to a 6.x or 7.x deployment target.

To map a composition unit to a deployment target, select a target from the **Available** list and click **>>**. The target name is displayed in the **Selected** list.

***Selected:***

Lists the names of desired deployment targets.

When you click **OK**, the product maps the composition unit to the deployment targets in the **Selected** list.

To remove a deployment target from the **Selected** list, select the target and click **<<**.

## Deployable unit relationship settings

Use this panel to specify relationship options for deployable units in an asset deployed as part of a business-level application. Specifying a relationship declares a dependency relationship that a deployable unit has on another asset deployed as a shared library in the same business-level application.

To view this administrative console panel, click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → ***deployed_asset_name*** → **Manage Relationships**. This help also pertains to both the Relationship options panel and the Composition unit relationship options panel that are shown when you add multiple deployable unit assets to a business-level application. These panels are shown for the **Define relationship with existing composition units** and **Options for creating new composition units to satisfy asset relationships** wizard steps.

A business-level application consists of composition units. When you add an asset to a business-level application, the product creates a composition unit for the asset. The composition unit name can be different from the name of the asset being deployed. The list of deployed assets shown for a business-level application consists of the composition unit names for the deployed assets. The relationships defined in this panel are composition unit relationships. The deployable units listed for a composition unit are those you chose from the associated asset when adding the asset. Composition unit relationships are expressed as deployable unit dependencies on other composition units belonging to the same business-level application. Only a composition unit for an asset deployed as a shared library can be specified as a dependency. You can map each deployable unit to a target independently from the others. Modifying relationships in this panel only affects the composition unit, not the associated asset.

To specify relationship options, select a deployable unit and click a button.

| Button | Resulting action |
| --- | --- |
| **Set Relationships** | Displays a panel through which you can add or change relationships for the deployable unit. Specify a relationship if a deployable unit depends on another asset deployed as a shared library in order to run.<br><br>This button is on the Set relationship options panel. |
| **Enable Match Targets** | If the deployable unit has a dependency relationship defined, click **Enable Match Targets** to map the related deployed assets to the same deployment targets as the dependent deployable unit. |

| Button | Resulting action |
| --- | --- |
| **Disable Match Targets** | If the deployable unit has a dependency relationship defined, click **Disable Match Targets** if the related deployed assets do not need to be deployed to the same targets as the deployable unit. |

***Deployable unit name:***

Specifies the name of the deployable unit of the selected deployed asset.

***Relationship:***

Specifies the composition unit names for all relationships defined for the associated deployable unit.

This setting is on the Set relationship options panel.

By default, a deployable unit has no relationships. To add or change related composition units, do the following:
1. Select the deployable unit.
2. Click **Set Relationships**.
3. Select the composition units that the deployable unit requires by moving them from the **Available** list to the **Selected** list.
4. Click **OK**.

***Match targets:***

Indicates the match targets value selected for the associated deployable unit. The default value is `true`.

A match targets value of `true` maps the composition units listed under **Relationship** to the same deployment targets as the associated deployable unit. Typically, you must deploy related composition units to the same targets as the dependent deployable unit in order for the deployable unit to run.

A `false` value indicates that the related composition unit can map to deployment targets which are different from the deployment targets of the deployable unit.

To set the value to `true`, select the deployable unit and click **Enable Match Targets**. To set the value to `false`, select the deployable unit and click **Disable Match Targets**. To set this value, the deployable unit must have a related composition unit.

## Business-level application settings

Use this page to configure a business-level application.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → ***application_name***.

This page is the same as the Adding composition unit to the business-level application page.

### Name

Specifies a logical name for the application. An application name must be unique within a cell and cannot contain an unsupported character.

An application name cannot begin with a period (.), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

| Unsupported characters | | |
|---|---|---|
| **/** forward slash | **$** dollar sign | **’** single quote mark |
| **** backslash | **=** equal sign | **″** double quote mark |
| ***** asterisk | **%** percent sign | **\|** vertical bar |
| **,** comma | **+** plus sign | **<** left angle bracket |
| **:** colon | **@** at sign | **>** right angle bracket |
| **;** semi-colon | **#** hash mark | **&** ampersand (and sign) |
| **?** question mark | **]]>** No specific name exists for this character combination | |

**Data type** String

## Description

Specifies a description for the business-level application.

## Deployed assets

Specifies the asset and shared library composition units in the business-level application. A *composition unit* is a registered asset or shared library that has additional configuration information, which you specify when adding the asset to the application.

For each composition unit, the table provides a name, description, asset type, and the runtime status of the composition unit.

| Button | Resulting action |
|---|---|
| **Add > Add Asset** | For assets that contain Java Platform, Enterprise Edition (Java EE) applications or modules, opens the application installation wizard. On the Select installation options panel of this wizard, you can specify a **Business-level application name** value that identifies the target business-level application. On the Map shared library relationships panel, you can identify the shared library files that individual modules need to run and specify composition unit names for the module-shared library relationships.<br><br>For non-Java EE assets, opens a wizard that helps you add an asset as a composition unit to your business-level application. |
| **Add > Add Shared Library** | Opens a wizard that helps you add a library file as a composition unit to your business-level application. |
| **Delete** | Deletes the composition unit from the product configuration repository and deletes the application binaries from the file system of all nodes where the application modules are installed.<br><br>On single-server installations, deletion occurs after the configuration is saved. |

## Business-level applications

Specifies the business-level applications in this business-level application.

The table provides a name, description, and the runtime status of each contained business-level application.

| Button | Resulting action |
|---|---|
| **Add** | Opens a wizard that helps you add a business-level application to your business-level application. |

| Button | Resulting action |
| --- | --- |
| **Delete** | Deletes the business-level application from the product configuration repository and deletes the application binaries from the file system of all nodes where the application modules are installed.<br><br>On single-server installations, deletion occurs after the configuration is saved. |

## Composition unit settings

Use this page to view composition unit settings and to change the deployment target of a composition unit.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → ***composition_unit_name***.

### Name

Specifies a logical name for the composition unit. You cannot change the name on this page.

### Description

Specifies a description for the composition unit. You cannot change the description on this page.

### Backing identifier

Specifies a unique identifier for a composition unit that is registered in the application management domain.

The identifier has the following format: `WebSphere:`*`unit_type`*`name=`*`unit_name`*. For example, for the MyApp.jar asset, the backing identifier might be `WebSphere:assetname=MyApp.jar`.

You cannot change the identifier on this page.

| | |
| --- | --- |
| **Data type** | String |
| **Units** | Configuration unit identifier |

### Starting weight

Specifies the order in which composition units are started when the server starts. The starting weight is like the startup order. The composition unit with the lowest starting weight is started first.

The value that you set for **Starting weight** determines the importance or weight of a composition unit within the business level application. For example, for the most important composition unit within a business-level application, specify `1` for **Starting weight**. For the next most important composition unit within the business-level application, specify `2` for **Starting weight**, and so on.

| | |
| --- | --- |
| **Data type** | Integer |
| **Default** | 1 |
| **Range** | 0 to 2147483647 |

### Start on distribution

Specifies whether to start the composition unit when the product distributes the composition unit to other locations.

The default is not to start the composition unit.

This setting applies to asset or shared library composition units. This setting does not apply when the composition unit is a business-level application.

| | |
| --- | --- |
| **Data type** | Boolean |

**Default** false

### Recycle behavior on update

Specifies whether the product restarts the composition unit after the composition unit is updated.

The default is to restart the composition unit after partial updating of the composition unit.

This setting applies to asset or shared library composition units. This setting does not apply when the composition unit is a business-level application.

| Option | Description |
|---|---|
| ALL | Restarts the composition unit after the entire composition unit is updated |
| DEFAULT | Restarts the composition unit after the part of the composition unit is updated |
| NONE | Does not restart the composition unit after the composition unit is updated |

### Current target

Specifies the existing deployment target for the composition unit.

To change the deployment target, click **Modify target** and select a different deployment target from the list of available clusters and servers.

## Example: Creating a business-level application

You can add many different types of artifacts to business-level applications. For example, you can add Java Platform, Enterprise Edition (Java EE) applications or modules, Java archives (JAR files), data in compressed files, and other business-level applications.

### About this task

An example of creating a simple business-level application follows. This example assumes that you have a compressed file, such as a zip file, or other archive available on your computer or on a remote server that you can use to complete the example.

If you do not have a compressed file available, look in product directories. Installing the product samples adds several sample files to the /samples directory. You can use these sample files in a business-level application.

1. Import assets.
   a. Click **Applications** → **New application** → **New Asset** in the console navigation tree.
   b. On the Upload asset page, specify the asset package to import and click **Next**.
      For example, specify a compressed file such as a zip file and click **Next**.
   c. On the Select options for importing an asset panel, click **Next**.
   d.
   e. On the Summary panel, click **Finish**.
   f. On the Adding asset to repository panel, if messages show that the operation completed, click **Manage assets**.
   g. On the Assets page, click **Save**.

   The file name displays in the list of assets.
2. Create an empty business-level application named `MySampleBLA`.
   a. Click **Applications** → **New application** → **New Business Level Application**.

b. On the New business-level application panel, specify a unique name such as `MySampleBLA` and a description, and then click **OK**.
c. On the business-level application settings page, click **Save**.

The name and description are shown in the list of applications on the Business-level applications page. Because the application is empty, its status is `Unavailable`.

3. Add the asset composition unit to your business-level application.
   a. On the Business-level applications page, click the application name in the list of applications.
   b. On the business-level application settings page, click **Add** → **Add Asset**.
   c. On the Add composition unit panel, select an asset composition unit from the list of available units, and then click **Continue**.

      For example, select the compressed file asset and then click **Continue**.
   d. On the Set options panel, click **Next**.
   e. On the Map composition unit to a target panel, change the target server as needed, and then click **Next**.
   f. On the Summary panel, click **Finish**. Several messages are displayed. A message having the format `Completed res=[WebSphere:cuname=`*`unit_name`*`]` indicates that the addition is successful.
   g. If the addition is successful, click **Manage application**.
   h. On the business-level application settings page, click **Save**.

   The asset name and type displays in the list of deployed assets.
4. Start the business-level application.
   a. Click **Applications** → **Application Types** → **Business-level applications**.
   b. On the Business-level applications page, select the check box beside your application.
   c. Click **Start**.

   When the business-level application is running, a green arrow displays for **Status**. If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again.

### What to do next

You can add other assets to your business-level application.

## Starting business-level applications

You can start a business-level application that is not running (has a status of `Stopped`). The application must contain code that can run on a server to start.

### Before you begin

The application must be installed on a server. By default, the application starts automatically when the server starts.

### About this task

You can start and stop business-level applications manually using the administrative console or wsadmin commands.

This topic describes how to use the administrative console to start a business-level application.

1. Go to the Business-level applications page.

   Click **Applications** → **Application Types** → **Business-level applications** in the console navigation tree.

2. Select the check box for the application you want started.
3. Click **Start**. The product runs the application and changes the state of the application to `Started`. The status is changed to `partially started` if not all servers on which the application is deployed are running.

### Results

A message stating that the application started displays at the top the page.

### What to do next

To restart a running application, select the application you want to restart, click **Stop** and then click **Start**.

## Stopping business-level applications

You can stop a business-level application that is running and has a status of `Started`).

### Before you begin

The application must be running on a product server.

### About this task

You can stop applications manually using the administrative console or wsadmin commands.

This topic describes how to use the administrative console to stop a business-level application.

1. Go to the Business-level applications page.

   Click **Applications** → **Application Types** → **Business-level applications** in the console navigation tree.
2. Select the check box for the application you want stopped.
3. Click **Stop**. The product stops the processing of the application and changes the state of the application to `Stopped`.

### Results

The status of the application changes and a message stating that the application stopped displays at the top the page.

### What to do next

To restart a stopped application, select the application you want to restart, and then click **Start**.

## Updating business-level applications

You can update business-level applications by deleting or changing composition units, or by mapping composition units to different deployment targets.

### Before you begin

Determine what changes that you want to make to your application. Also, determine whether the changed application can run on your deployment targets.

## About this task

Updating consists of adding new composition units to an application, replacing or removing composition units, or mapping composition units to different deployment targets.

You can add an asset or shared library composition unit to multiple business-level applications. However, each composition unit for the same asset must have a unique composition unit name. You can add a business-level application composition unit to more than one business-level application.

This topic describes how to update business-level applications using the administrative console. Alternatively, you can use programming or the wsadmin tool.

- Delete composition units from your business-level application.
  1. Go to the business-level application settings page.

     Click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** in the console navigation tree.
  2. Select each composition unit of the application that you want to delete.
  3. Click **Delete**.
  4. On the Delete composition unit from business-level application panel, confirm the deletion and click **OK**.
- Add new or updated assets, shared libraries, or other business-level applications to your business-level application.
  1. Update asset binary files or shared libraries as needed.
  2. If you are adding new assets that are not registered with the product management domain, import the assets.
  3. If you are updating existing assets, use the **Update** option to update asset files.
  4. On the business-level application settings page, specify the type of composition unit to add.
     - To add an asset, under **Deployed assets**, click **Add** → **Add Asset**.
     - To add a shared library, under **Deployed assets**, click **Add** → **Add Shared Library**.
     - To add a business-level application, under **Business-level applications**, click **Add**.
  5. On the New composition unit panel, select a unit from the list of available units, and then click **Continue**.
  6. On the Set options panel, change the composition unit settings as needed, and then click **Next**.
  7. On the Map composition unit to a target panel, change the deployment target as needed, and then click **Next**.

     This panel is not shown when you add a business-level application.
  8. On the Summary page, click **Finish**.
  9. If the product adds the unit successfully, click **Manage application**.

     If the unit addition is not successful, read the messages, and try adding the unit again. Correct the errors noted in any messages.
  10. On the Adding composition unit to the business-level application page, click **Save**.
  11. Repeat these steps to add any other assets, shared libraries, or applications needed by the business-level application.

  The business-level application settings page displays the configuration unit names.
- Map composition units to different deployment targets.
  1. On the composition unit settings page, select the composition unit that you want to change.
  2. Under **Current targets**, click **Modify Target**.
  3. On the Map targets page, change the target.
     a. From the list of available clusters and servers, select a different deployment target.
     b. Click **>>** to add the deployment target to the **Selected** list.

c. To remove a deployment target from the **Selected** list, select the target and click **<<**.
d. Click **OK**.

The business-level application settings page displays the selected deployment target.

### What to do next

Save the changes to your administrative configuration.

## Deleting business-level applications

After an application no longer is needed, you can delete it.

### About this task

Deleting a business-level application removes the application from the product configuration repository and it deletes the application binaries from the file system of all nodes where the application files are installed.

1. Go to the Business-level applications page.

   Click **Applications** → **Application Types** → **Business-level applications** in the console navigation tree.
2. If you need to retain a copy of the application, back up composition units of the application.
3. Delete composition units of the application.
   a. On the Business-level applications page, click the name of the business-level application that you want to delete.
   b. On the business-level application settings page, select each composition unit of the application.
   c. Click **Delete**.
   d. On the Delete composition unit from Business-level application panel, confirm the deletion and click **OK**.

   Deleting a configuration unit removes the configuration from the *profile_root*/config/cells/*cell_name*/cus directory.
4. Delete the business-level application.
   a. Select the application that you want to delete.
   b. Click **Delete**.

   Unless the application is used by another business-level application, deleting a business-level application removes the configuration from the *profile_root*/config/cells/*cell_name*/blas directory.
5. On the Delete business-level application panel, confirm the deletion and click **OK**.
6. Save changes made to the administrative configuration.

### Results

On single-server products, application binaries are deleted after you save the changes.

### What to do next

If using the administrative console **Delete** options does not fully delete a business-level application or its configuration units, you can delete the business-level application and its configuration units manually from a stand-alone server. Suppose you want to delete a business-level application named ExampleBLA, and ExampleBLA is not used by another business-level application. Complete the following steps to manually delete the ExampleBLA configurations from the blas and cus directories:

1. Delete the *profile_root*/config/cells/*cell_name*/blas/ExampleBLA directory.
2. Delete the *profile_root*/config/cells/*cell_name*/cus/ExampleBLA directory.

3. Save changes made to the administrative configuration.

# Chapter 8. Administering business-level applications using programming

You can use the command framework programming to create, edit, update, start, stop, delete, export, import, and query information about business-level applications. A business-level application defines an enterprise-level application.

## Before you begin

This task assumes a basic familiarity with the command framework. Read about the command framework in the application programming interfaces documentation.

## About this task

Besides creating, editing, updating, starting, stopping, deleting, exporting, importing, and querying information about business-level applications using programming, you can do these tasks using the administrative console or the wsadmin scripting tool.

1. Perform any of the following tasks to administer your business-level applications using programming.
   a. Create an empty business-level application.

      You typically create an empty business-level application and then add assets or business-level applications as composition units to the empty business-level application.
   b. Import an asset.

      You can import an asset to register the asset with the product and optionally store the asset in the product repository so that you can later use the asset in a business-level application. An asset represents at least one binary file that implements business logic.
   c. Add a composition unit.

      You can add an asset to a business-level application by creating a composition unit for the asset. A composition unit is typically created from an asset and contains configuration information that makes the asset runnable.
   d. Start a business-level application.

      You can start a business-level application, which starts each composition unit in that business-level application. Each composition unit is started on the respective targets on which the business-level application is deployed.
   e. Stop a business-level application.

      You can stop a business-level application, which stops each composition unit in that business-level application. Each composition unit is stopped on the respective targets on which the business-level application is deployed.
   f. Check the status of a business-level application.

      You can check the status of an entire business-level application. You can also limit the status to a particular composition unit of a business-level application, a specific deployment target, or check the status of the composition unit and the deployment target at the same time.
   g. Delete a business-level application.

      You can delete a business-level application using programming. You might delete a business-level application if the application is not functioning correctly, no longer needed, and so on.
   h. Delete an asset.

      You can delete an asset from a business-level application using programming if the asset is not functioning corrctly, the asset is no longer needed, and so on. An asset represents at least one binary file that implements business logic.
   i. Delete a composition unit.

You can delete a composition unit from a business-level application if the composition unit is not functioning correctly, the composition unit is no longer needed, and so on. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

j. Export an asset.

You can export an asset from the current session so that you can back up the asset, import the asset to another session, and so on. An asset represents at least one binary file that implements business logic.

k. List assets.

You can list the assets that have been imported to the current workspace so that you can do further asset administration, such as deleting or exporting assets. An asset represents at least one binary file that implements business logic.

l. List composition units.

You can list the composition units for a specific business-level application in a session so that you can do further composition unit administration, such as deleting or adding composition units. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

m. List business-level applications.

You can list the business-level applications of a session so that you can do further business-level application administration such as deleting a business-level application. A business-level application is an administrative model that captures the definition of an enterprise-level application so that you can perform specific business functions, such as accounting.

n. Edit a composition unit.

You can edit the configuration information in a composition unit of a business-level application if, for example, you want to change which modules in the composition unit are configured to run in which targets. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

o. Edit an asset.

You can edit the information of an asset, for example, its destination location, its relationship with other assets, and so on. An asset represents at least one binary file that implements business logic

p. Edit a business-level application.

You can edit the information of a business-level application such as its description. A business-level application is an administrative model that captures the entire definition of an enterprise-level application.

q. Update an asset.

You can update an asset by adding, deleting, or updating a single file or Java Platform, Enterprise Edition (Java EE) module, or by merging multiple files or Java EE modules into an asset. You can also update an asset by replacing the entire asset.

r. View a composition unit.

You can view the composition unit information so that you can do other tasks associated with the composition unit, such as editing an asset or deleting a composition unit. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

s. View an asset.

You can view the asset information so that you can do other tasks associated with the asset, such as editing or exporting an asset. An asset represents at least one binary file that implements business logic.

t. View a business-level application.

You can view business-level application information such as the description so that you can do other tasks associated with the business-level application, such as editing the business-level application. A business-level application is an administrative model that captures the entire definition of an enterprise-level application.

u. List control operations.

You can list the control operations of a business-level application or a composition unit for a session. You use control operations, such as start or stop, to change or query the runtime environment of a business-level application or a composition unit.

2. Save your changes to the master configuration repository.
3. Synchronize changes to the master configuration across the nodes for the changes to take effect.

### Results

Depending on which tasks you complete, you have created, edited, updated, started, stopped, deleted, exported, imported, or queried information about business-level applications.

### What to do next

If you have further business-level application updates, you can do the updates through programming, the administrative console, or the wsadmin scripting tool.

## Creating an empty business-level application using programming

You can create an empty business-level application, and then add assets or business-level applications as composition units to the empty business-level application.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

You can create an empty business-level application using programming, the administrative console, or the wsadmin tool.

### About this task

Perform the following steps to create an empty business-level application using programming. In your code that creates the empty business-level application, you must provide the name parameter. The name parameter specifies the name of the business-level application that you create.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is

passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Use the command manager that you created in a previous step to create and set up the command that creates an empty business-level application.

   The command name is createEmptyBLA. The name parameter is a required parameter that you use to specify the name of the business-level application. You can optionally provide the description parameter to provide a description of the newly created business-level application.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the execute method in the asynchronous command client to run the command that creates an empty business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the empty business-level application is created.

## Example

The following example shows how to create an empty business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.exception.AdminException;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class CreateEmptyBLA {

    public static void main(String[] args) {

        try {

        // Connect to the application server.
        // This step is optional if you use the local
        // command manager. Comment out the lines to and including
        // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
        // soapClient);
        // to get the soapClient soap client if you use the local
        // command manager.

        String host = "localhost";
        String port = "8880";
        // Change to your port number if it is
        // not 8880.
```

```
Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
        AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
              AdminClientFactory.createAdminClient(config);

// Create the command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler
// for listening to command notifications.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the AsyncCommandClient object that follows.

AsyncCommandClient asyncCmdClientHelper = new
             AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that creates an empty
// business-level application.
String cmdName = "createEmptyBLA";

AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Create an empty
              // business-level application using
              // the session created.
System.out.println("\nCreated " + cmdName);

// Set the name command parameter.
String blaName = "bla1";
cmd.setParameter("name", blaName);

System.out.println("\nSet name parameter to "
                    + cmd.getParameter("name"));

// Uncomment the following lines to set the description of
// the business-level application being created:
//
// String blaDescription = "description for bla1";
// cmd.setParameter("description", blaDescription);
// System.out.println("\nSet description parameter to " +
//                      cmd.getParameter("description"));

// Call the asynchronous command client to
```

```
        // process the command parameters.
        try {
             asyncCmdClientHelper.processCommandParameters(cmd);
             System.out.println("\nCompleted process command " +
                                "parameters");
        } catch (Throwable th) {
            System.out.println("Failed from " +
                "asyncCmdClientHelper.processCommandParameters(cmd).");
            th.printStackTrace();
            System.exit(-1);
        }

        // Call the asynchronous command client to run the command.
        asyncCmdClientHelper.execute(cmd);
        System.out.println("\nCompleted command execution");

        // Check the command result.
        CommandResult result = cmd.getCommandResult();
        if (result != null) {
             if (result.isSuccessful()) {
                System.out.println("\nCommand ran successfully "
                       + "with result\n" + result.getResult());
             } else {
                System.out.println("\nCommand ran with " +
                                    "Exception");
                result.getException().printStackTrace();
             }
        }
     } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification.
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

You can add business-level applications or assets as composition units into the newly created business-level application. Alternatively, you can add the newly created business-level application to other business-level applications.

## Importing an asset using programming

You can import an asset to register the asset with the product and optionally store the asset in the product repository so that you can later use that asset in a business-level application. An asset represents at least one binary file that implements business logic.

## Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

You can import an asset using programming, the administrative console, or the wsadmin tool.

## About this task

When you import an asset, you register the asset with the product and optionally store the asset in the product repository.

You must provide a file path to the source that you are importing. Specify an absolute path name to the source, as the behavior for a relative path is unpredictable.

You can specify a destination location from where the application server reads the asset file while starting a composition unit created from the asset. The asset is copied to this location when the configuration session is saved after the asset is imported. The default asset destination is *profile_root*/installedAssets/ *asset_name*.

You can optionally specify a storage type of FULL, METADATA, or NONE. The default value is FULL, which means that the asset and associated meta data are stored in the product asset repository. If you specify a storage type of METADATA, the asset is not copied to the product repository, but associated meta data is stored in the product repository. If you specify a storage type of NONE, neither the asset nor the asset meta data is stored in the product asset repository. For storage types of METADATA and NONE, the asset is expected to reside at the destination file path. Storage types of METADATA and NONE are typically used by development tools which enable iterative development on the copy of the asset in the directory structure of the tool.

Perform the following steps to import an asset using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to set up the command that imports an asset.

   The command name is importAsset. The source parameter is a required parameter that you use to specify the path to the asset. You can optionally provide the storageType parameter to specify how to save the asset in the configuration repository.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Set up the command step parameters.

   You can set parameters in the AssetOptions step that contains data about the asset such as its description, file permission, and relationship with other assets.
8. Call the asynchronous command client to run the command that imports an asset.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
9. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the asset is imported.

## Example

The following example shows how to import an asset based on the previous steps.

Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;
import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;

import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.UploadFile;
import com.ibm.websphere.management.exception.AdminException;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ImportAsset {

    public static void main (String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.

            String host = "localhost";
            String port = "8880"; //Change to your port number if it is not
                                  //8880.

            Properties config = new Properties();
            config.put (AdminClient.CONNECTOR_HOST,  host);
            config.put (AdminClient.CONNECTOR_PORT,  port);
            config.put (AdminClient.CONNECTOR_TYPE,
                        AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println ("Config: " + config);
```

```
            AdminClient soapClient =
                          AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

            // Comment out the previous lines to create a client command
            // manager if you are using a local command manager.
            // Uncomment the following line to create a local command
            // manager:
            //
            //CommandMgr cmdMgr = CommandMgr.getCommandMgr();

            System.out.println("\nCreated command manager");

            // Optionally create the asynchronous command handler.
            // Comment out the following line if no further handling
            // of command notification is required:
            AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

            // Create an asynchronous command client.

            // Setup the session.
            String id = Long.toHexString(System.currentTimeMillis());
            String user = "content" + id;
            Session session = new Session(user, true);

            // If no command handler is used, replace the following listener with
            // null for the AsyncCommandClient object.

            AsyncCommandClient asyncCmdClientHelper = new
                        AsyncCommandClient(session, listener);
            System.out.println("\nCreated async command client");

            String cmdName = "importAsset";
            UploadFile assetSource = new
UploadFile("/sources/test5.zip");  //Change to the directory of your sources.

            // Create the command to import an asset.
            AdminCommand cmd = cmdMgr.createCommand(cmdName);
            cmd.setConfigSession(session); //import the asset using
                                           //the session created
            System.out.println("\nCreated " + cmdName);

            // Set the source command parameter.
            cmd.setParameter("source", assetSource);
            System.out.println("\nSet source parameter to " +
                               cmd.getParameter("source"));

            // Uncomment the following line to set the storage type to
            // a value of STORAGETYPE_META or STORAGETYPE_NONE instead of
            // the default of STORAGETYE_FULL:
            //
            //cmd.setParameter("storageType,
            //                  CommandConstants.STORAGETYPE_NONE);

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                   "parameters");
            }
            catch(Throwable th) {
                System.out.println("Failed from " +
                     "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
```

```
    System.exit(-1);
}

// Set up the step parameters for the AssetOptions step.
String stepName = "AssetOptions";
CommandStep step = ((TaskCommand) cmd).gotoStep(stepName);

// The new asset name must contain the
// same extension as the original .zip file name.
String assetNewName = "asset1.zip";

// If you override the default destination, include the
// entire path with the file name for the new destination.
String destName = "/websphere/asset/installDir/asset1.zip";
for (int i = 0; i < step.getNumberOfRows(); i++) {
    // The following lines change the name and destination
    // step parameters. Other step parameters that you can
    // use follow, but are commented out.
    // Change your set
    // of step parameters as required by your scenario.

    // Set the name.
    step.setParameter("name", assetNewName, i);
    System.out.println("\nSet name parameter to " +
                       step.getParameter("name", i));

    // Set the destination.
    step.setParameter("destination", destName, i);
    System.out.println("\nSet destination parameter to " +
                       step.getParameter("destination", i));

    // Set the description.
    //String desc = "description for asset1.zip";
    //step.setParameter("description", desc, i);
    //System.out.println("\nSet description parameter to " +
    //     step.getParameter("description", i));

    // Set the validation.
    //String validate = "Yes";
    //step.setParameter("validate", validate, i);
    //System.out.println("\nSet validate parameter to " +
    //            step.getParameter("validate", i));

    // Set the file permission.
    //String filePermission = ".*\\.dll=755";
    //step.setParameter("filePermission", filePermission, i);
    //System.out.println("\nSet filePermission parameter to " +
    //            step.getParameter("filePermission", i));

    // Set the type aspect parameter value.
    // Format for a typeAspect: WebSphere:spec=xxx,version=n.n+
    // Websphere:spec=xxx,version=n.n.
    //String typeAspect = "";
    //step.setParameter("typeAspect", typeAspect, i);
    //System.out.println("\nGet typeAspect: " +
    //     step.getParameter("typeAspect", i));

    // Set the relationship parameter.
    // The relationship parameter declares dependency
    // relationships on other assets.  The parameter value
    // is a list which contains the ID of each asset declared
    // as a dependency.  Each ID in the list is separated by
    // a "plus" sign ("+").
    //
    // Only assets which are Java archives can be referenced in
    // dependency relationships.  An asset is a Java archive if
```

```
                // it has a type aspect identifying it as such.
                //
                // If an asset declared as a dependency does not exist or
                // does not have a Java archive type aspect, it is ignored
                // and no dependency on the asset is registered in the
                // asset's configuration.
                //
                //String relationship =
                //       "assetname=shared.zip+assetname=shared2.zip";
                //step.setParameter("relationship", relationship, i);
                //System.out.println("\nGet relationship: " +
                //      step.getParameter("relationship", i));

            }

            // Call the asynchronous command client that imports the asset.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully " +
                           "with result\n" + result.getResult());
                } else {
                    System.out.println("\nCommand ran with " +
                                       "exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

Add a composition unit to a business-level application using the asset that you imported. An asset included in a business-level application is represented by a composition unit.

## Adding a composition unit using programming

You can add an asset to a business-level application by creating a composition unit for the asset. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

## Before you begin

Before you can add a composition unit to a business-level application, you must have created an empty business-level application and imported an asset.

You can add a composition unit to a business-level application using programming, the administrative console, or the wsadmin tool.

## About this task

When you add a composition to a business-level application, the composition unit is configured for the specified business-level application. The composition unit cannot be shared with other business-level applications.

Perform the following steps to add a composition unit to a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that adds a composition unit.

   The command name is addCompUnit. The blaID and cuSourceID parameters are required parameters that you use to specify composition unit source to be added to the business-level application. Examples of composition unit source are an asset or a business-level application. You can optionally provide deployable units for the composition unit through the deplUnit parameter. If the cuSourceID parameter is a Java Platform, Enterprise Edition (Java EE) asset, you can optionally use the cuConfigStrategyFile parameter or the defaultBindingOptions parameter to specify the default bindings. The defaultBindingOptions parameter must match the binding options available for this Java EE asset. To view a list of binding options available for this Java EE asset, look at the AssetOptions step in the viewAsset command. Specify each binding option in an *option_name*=*option_value* pair, with multiple pairs separated by a # character.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Set up the command step parameters.

   You can set up composition unit information through various steps. The CUOptions step contains data about the composition unit such as its description, starting weight, and start and restart behavior. The MapTargets step contains target information about where the composition unit is to be deployed. The RelationshipOptions step contains shared library composition units on which this composition unit has dependencies. The ActivationPlanOptions step allows you to specify runtime components for each

deployable unit. The CreateAuxCUOptions step contains assets on which this composition unit has dependencies. You can set up parameters in these steps.

8. Call the asynchronous command client to run the command that adds a composition unit to a business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

9. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the composition unit is added to the business-level application.

## Example

The following example shows how to import an asset based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class AddCompUnit {

    public static void main(String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local command
            // manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the
            // local command manager.

            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                       AdminClientFactory.createAdminClient(config);

            // Create the command manager
```

```
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();

System.out.println("\nCreated command manager");

// Optionally create the asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the following listener with
// null for the AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command to add a composition unit to a business-level application.
String cmdName = "addCompUnit";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Add the composition unit using
                              // the session created.
System.out.println("\nCreated " + cmdName);

// Set the blaID command parameter.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an
// incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                  + cmd.getParameter("blaID"));

// Set the cuSourceID command parameter.
// Examples of valid formats for the cuSourceID parameter:
// If the source is an asset, examples are:
// - aName
// - assetname=aName
// - WebSphere:assetname=aName
// If the source is another business-level application,
// examples are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// The cuSourceID command parameter
// accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique asset or business-level application.
String cuSourceID = "assetname=asset1.zip";
cmd.setParameter("cuSourceID", cuSourceID);
```

```
System.out.println("\nSet cuSourceID parameter to "
                   + cmd.getParameter("cuSourceID"));

// Set the deplUnits command parameter.
// If the deployable units of an asset are, for example, a.jar and
// b.jar, then when you run the addCompUnit command you can
// specify deplUnits as a.jar+b.jar. You can specify the whole
// list, a subset of that list, or "default" to create this composition
// unit as a shared library. If the deplUnits parameter is not specified,
// the deployable units are set the same as that of their asset.
String deplUnits = "default";
cmd.setParameter("deplUnits", deplUnits);

System.out.println("\nSet deplUnits parameter to "
                 + cmd.getParameter("deplUnits"));

// Call the asynchronous client helper to process parameters.
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
    System.out.println("\nCompleted process command " +
                           "parameters");
} catch (Throwable th) {
    System.out.println("Failed from " +
        "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}

// Set up the step parameters for the CUOptions step.
// The CUOptions step contains the following arguments:
// description - description for the composition unit
// startingWeight - starting weight for the composition
// unit within the business-level application. The default is 1.
// startedOnDistributed - to start composition unit upon distribution
// to target nodes. The default is false.
// restartBehaviorOnUpdate - restart behavior for a composition unit when
// updating the composition unit.
// The default is DEFAULT. Valid values are DEFAULT, ALL, and NONE.
String stepName = "CUOptions";
CommandStep step = ((TaskCommand) cmd).gotoStep(stepName);

// Composition unit name:
String name = "cu1";

// Composition unit description:
String description = "cu1 description";

for(int i = 0; i < step.getNumberOfRows(); i++) {
   // The following lines change the composition unit name and
   // description step parameters of the CUOptions step. Change
   // your set of step parameters as required for your
   // scenario.

   // Set the name.
   step.setParameter("name", name, i);
   System.out.println("\nSet name parameter to " +
                      step.getParameter("name", i));

   // Set the description.
   step.setParameter("description", description, i);
   System.out.println("\nSet description parameter to " +
                      step.getParameter("description", i));
}

// Set up the step parameters for the MapTargets step.
stepName = "MapTargets";
step = ((TaskCommand) cmd).gotoStep(stepName);
```

```
// Specify the targets to deploy the composition unit.
// The default is server1. Use the + character to
// specify multiple targets.
String server = "server1";

for(int i = 0; i < step.getNumberOfRows(); i++) {
    // The following lines change the composition unit and
    // server step parameters of the
    // MapTargets step. Change your set of step parameters
    // as required for your scenario.

    // Set the server.
    step.setParameter("server", server, i);
    System.out.println("\nSet server parameter to " +
                            step.getParameter("server", i));
}

// The addCompUnit command might contain the
// CreateAuxCUOptions, RelationshipOptions and ActivationPlanOptions
// steps, depending on the asset content of the assets imported.
// The CreateAuxCUOptions step is available if the cuSourceID value
// is an asset. The asset includes an asset relationship to an
// asset that does not have a matching composition unit in the
// business-level application.
//
// If the CreateAuxCUOptions step is available, the selected
// deployable units of the source asset of the "primary" composition
// unit (that is, the composition unit being added) have dependencies
// on other assets for which there are no matching composition units
// in the business-level application. A "secondary" composition unit will be created for each
// of those asset dependencies.
//
// Each CreateAuxCUOptions row corresponds to one dependency
// relationship declaration.  Each row consists of parameter values
// for the dependency relationship.  Some parameters are read-only and
// some of them are editable.  To edit parameter values, use the same
// approach as that used to edit parameter values in the CUOptions step.
//
// The parameters for this step include:
//
//  deplUnit - The name of the deployable unit which has the
//             dependency. (Read-only.)
//  inputAsset - The asset ID for the source asset of the primary
//               composition unit. (Read-only.)
//  cuID - The name of the secondary composition unit to create.
//  matchTarget - Specifies whether the server target for the secondary
//                composition unit is to match the server target for
//                the primary composition unit.  The default value
//                is "true".  If the value is set to "false", the
//                secondary composition unit will be created with no
//                target.  The target on the secondary composition unit
//                can be set at a later time with the editCompUnit
//                command.
//
// If the RelationshipOptions step is available, the selected
// deployable units of the source asset of the "primary" composition
// unit (that is, the composition unit being added) have dependencies
// on other assets for which there are matching "secondary" composition
// units in the business-level application.  The RelationshipOptions step is much like
// CreateAuxCUOptions except that the required secondary composition
// units already exist.  Also, each RelationshipOptions row maps one
// deployable unit to one or more secondary composition units, whereas,
// each CreateAuxCUOptions row maps one deployable unit to one
// asset dependency.
//
// Each RelationshipOptions row corresponds to one deployable unit
```

```
            // with one or more dependency relationships and consists of
            // parameter values for the dependency relationships.  Some parameters
            // are read-only and some of them are editable.  To edit parameter
            // values, use the same approach as that used to edit parameter values
            // in the CUOptions step.
            //
            // The parameters for this step include:
            //
            //  deplUnit - The name of the deployable unit which has the
            //             dependency. (Read-only.)
            //  relationship - Composition unit dependencies in the form of a
            //                 list of composition unit IDs.  Composition unit
            //                 IDs are separated by a "plus" sign ("+").  Each ID
            //                 can be fully or partially formed as shown with the
            //                 following examples:
            //                     WebSphere:cuname=SharedLib1.jar
            //                     WebSphere:cuname=SharedLib.jar
            //                     SharedLib.jar
            //  matchTarget - Specifies whether the server target for the secondary
            //                composition units are to match the server target for
            //                the primary composition unit.  The default value
            //                is "true".  If the value is set to "false", the
            //                secondary composition unit will be created with no
            //                target.  The target on the secondary composition unit
            //                can be set at a later time with the editCompUnit
            //                command.
            // The addCompUnit command contains the ActivationPlanOptions step.
            // The user can set the ActivationPlanOptions step parameters
            // similar to the step parameters for the CUOptions step in
            // the previous examples. The arguments for this step include:
            // deplUnit – deployable unit URI (read only parameter)
            // activationPlan - specifies a list of runtime components in the
            //                  format of specname=xxxx
            //
            // Run the command to add the composition unit.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
              if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                + "with result\n" + result.getResult());
               }
               else {
                   System.out.println("\nCommand ran with " +
                                            "Exception");
                   result.getException().printStackTrace();
               }
             }
         } catch (Exception e) {
             e.printStackTrace();
         }
     }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
```

```
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

Start the business-level application to which you added the composition unit. Complete administrative tasks such as viewing or deleting the composition unit.

## Starting a business-level application using programming

You can start a business-level application, which starts each composition unit in that business-level application. Each composition unit is started on the respective targets on which the business-level application is deployed.

### Before you begin

Before you can start a business-level application, you must have created an empty business-level application, imported an asset, and added a composition unit to the business-level application.

You can start a business-level application using programming, the administrative console, or the wsadmin tool.

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

### About this task

You must specify the blaID parameter of the business-level application that you start.

Perform the following steps to start a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to set up the command that starts a business-level application.

   The command name is startBLA. The blaID parameter is a required parameter to specify the business-level application to start.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to run the command that starts a business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the business-level application is started.

## Example

The following example shows how to start a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class startBLA {

    public static void main(String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local command
            // manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if
            // you use the local command manager.

            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                             AdminClientFactory.createAdminClient(config);

            // Create the command manager
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);
```

```
// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler
// for listening to command notifications.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the AsyncCommandClient object that follows.
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that starts the business-level application.
String cmdName = "startBLA";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Start a business-level
                         //application using the session created.
System.out.println("\nCreated " + cmdName);

// (required) Set the blaID parameter.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter
// accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                 + cmd.getParameter("blaID"));

// Call asynchronous client helper to process parameters.
try {
   asyncCmdClientHelper.processCommandParameters(cmd);
   System.out.println("\nCompleted process command " +
                            "parameters");
} catch (Throwable th) {
   System.out.println("Failed from " +
       "asyncCmdClientHelper.processCommandParameters(cmd).");
   th.printStackTrace();
   System.exit(-1);
}

// Call the asynchronous command client to run the command command.
asyncCmdClientHelper.execute(cmd);
System.out.println("\nCompleted running the command");

// Check the command result.
CommandResult result = cmd.getCommandResult();
if (result != null) {
```

```
            if (result.isSuccessful()) {
                System.out.println("\nCommand ran successfully "
                               + "with result\n" + result.getResult());
            }
            else {
                System.out.println("\nCommand ran with " +
                                          "Exception");
                result.getException().printStackTrace();
            }
         }
     } catch (Exception e) {
        e.printStackTrace();
     }
  }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

### What to do next

Your users can access the business-level application that you started.

## Stopping a business-level application using programming

You can stop a business-level application, which stops each composition unit in that business-level application. Each composition unit is stopped on the respective targets on which the business-level application is deployed.

### Before you begin

Before you can stop a business-level application, you must have created an empty business-level application, imported an asset, added a composition unit to the business-level application, and started the business-level application.

### About this task

You can stop a business-level application using programming, the administrative console, or the wsadmin tool. This topic describes how to stop a business-level application using programming.

Perform the following steps to stop a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.

4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Use the command manager that you created in a previous step Create and set up the command that stops a business-level application.

   The command name is stopBLA. The blaID parameter is a required parameter to specify the business-level application to stop.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to run the command that stops a business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the business-level application is stopped.

## Example

The following example shows how to stop a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class stopBLA {

    public static void main(String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local command
            // manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
```

```
// to get the soapClient soap client if
// you use the local command manager.

String host = "localhost";
String port = "8880"; // Change to your port number if it is
                      // not 8880.

Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
           AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
                  AdminClientFactory.createAdminClient(config);

// Create the command manager
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manage:.
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler
// for listening to command notifications.
// Comment out the following line if no further handling
// of command notification is required.
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the AsyncCommandClient object that follows.
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that stops the business-level application.
String cmdName = "stopBLA";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Stop a business-level
     //application that is using the session created.
System.out.println("\nCreated " + cmdName);

// (required) Set the blaID parameter.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter
// accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                    + cmd.getParameter("blaID"));
```

```
            // Call asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                           "parameters");
            } catch (Throwable th) {
                System.out.println("Failed from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Call the asynchronous command client to run the command command.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                                   "Exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

Complete administrative tasks on the business-level application, such as editing an asset or a composition unit that is contained in the business-level application.

## Checking the status of a business-level application using programming

You can check the status of an entire business-level application. You can also limit the status to a particular composition unit of a business-level application, a specific deployment target, or check the status of the composition unit and the deployment target at the same time.

## Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interface documentation.

Before you can check the status of a business-level application or a composition unit, you must have created the business-level application.

You can check the status of a business-level application using programming, the administrative console, or the wsadmin tool.

## About this task

You must provide the blaID parameter to specify the business-level application that you are viewing.

Perform the following tasks to view a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Create and set up the getBLAStatus command to check the status of a business-level application.
   a. Set the blaID parameter for the business-level application whose status you want to check.
   b. Optionally set the cuID parameter if you want to narrow the scope of the query to a single composition unit.
   c. Optionally set the targetID if you want to narrow the scope of the query to a single target server process or cluster.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to run the command to check the status of the business-level application.

   You could have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, you can check the status of an entire business-level application, if you chose not to limit the status. If you chose options to limit the status, you could check the status to a particular composition unit of a business-level application, a specific deployment target, or check the status of the composition unit and the deployment target at the same time.

The smallest unit of status data that the system maintains is for a single composition unit in a single server or cluster member process. Business-level application status can be based on one or more composition units, each having one or more targets, with targets potentially consisting of clusters with multiple member processes. Therefore, the single status value returned from the getBLAStatus command is a compilation of individual status data for all composition units on all target process within the scope of the status query. The following table describes how individual status data is compiled into a single status value. The term *composition unit instance* used in the table refers to a composition unit on a single server or single cluster member process.

| Status | Description |
| --- | --- |
| ExecutionState.STARTED | All composition unit instances within the scope of the query have been started. |
| ExecutionState.STOPPED | All composition unit instances within the scope of the query have not been started or have been stopped. |
| ExecutionState.PARTIAL_START | Some composition unit instances within the scope of the query have a status of ExecutionState.STARTED and some have a status of ExecutionState.STOPPED. |
| ExecutionState.UNKNOWN | Status data for at least one composition instance within the scope of the query cannot be obtained for some reason. |

## Example

The following example shows how to check the status of a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class EditBLA {

    public static void main(String[] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local command manager.
            // Comment out the following lines to get the soapClient soap client if
            // you are going to use the local command manager. You would
            // comment out the lines to and including
            // CommandMgr cmdMgr =
            // CommandMgr.getClientCommandMgr(soapClient);
```

```
String host = "localhost"; // Change to your host if it is not localhost.
String port = "8880"; // Change to your port number if it is not 8880.

Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
       AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
                  AdminClientFactory.createAdminClient(config);

// Create command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager.
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();

System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required.
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
// This example creates a new session. You can replace the
// following code to use an existing session that has been
// created.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the following AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
                AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command.
String cmdName = "getBLAStatus";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Check the status
                               // using the session
                               // created
System.out.println("\nCreated " + cmdName);

// Set the required blaID parameter
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
String blaID = "MyBLA";   // Replace the MyBLA value with your
                          // blaID value.
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
        + cmd.getParameter("blaID"));

// Optionally set the cuID parameter.
```

```
            String cuID = "myCU.zip";   // Replace the myCU.zip value with your
                                        // cuID value.
            cmd.setParameter("cuID", cuID);

            System.out.println("\nSet cuID parameter to "
                    + cmd.getParameter("cuID"));

            // Optionally set the targetID parameter.
            // The format of the targetID parameter for a cluster
            // is WebSphere:cluster=cluster1
            String targetID = "WebSphere:node=node1,server=server1";   // Replace
                                                // this with your targetID value.
            cmd.setParameter("targetID", targetID);

            System.out.println("\nSet targetID parameter to "
                    + cmd.getParameter("targetID"));

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted processCommandParameters");
            } catch (Throwable th) {
                System.out.println("Throwing an exception from " +
                   "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Run the command to check the status of the
            // business-level application.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted command execution");

            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand executed successfully "
                           + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand executed with " +
                                           "Exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification.
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

## What to do next

You can use the results of the status check to perform other tasks. For instance, if the results indicate that none of the composition units is started, you could start the business-level application.

# Deleting a business-level application using programming

You can delete a business-level application using programming. You might delete a business-level application if it is not functioning correctly, it is no longer needed, and so on.

## Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can delete a business-level application, you must have created an empty business-level application. You can optionally have added assets or business-level applications as composition units to the empty business-level application. All the composition units in the business-level application must be deleted using the deleteCompUnit command before you delete the business-level application. Other business-level applications cannot reference the business-level application that you are deleting. Otherwise, the deletion fails.

You can delete a business-level application using programming, the administrative console, or the wsadmin tool.

## About this task

You must specify the blaID parameter of the business-level application that you are deleting. You might delete a business-level application if it is not functioning correctly, no longer needed, and so on.

Perform the following steps to delete a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager you created in a previous step to create and set up the command that deletes the business-level application.

   The command name is DeleteBLA. The blaID parameter is a required parameter to specify the business-level application to delete.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to run the command that deletes the business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the business-level application is deleted.

## Example

The following example shows how to delete a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class DeleteBLA {

    public static void main(String [] args) {

        try {


            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.

            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                             AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

            // Comment out the previous lines to create a client command
```

```
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Setup the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace listener with
// null for the following AsyncCommandClient object:
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that deletes the business-level application.
String cmdName = "deleteBLA";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Delete the business-level
// application using the session created.
System.out.println("\nCreated " + cmdName);

// Set the blaID parameter to the business-level application to delete.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                    + cmd.getParameter("blaID"));


// Call the asynchronous client helper to process parameters.
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
    System.out.println("\nCompleted process command " +
                           "parameters");
} catch (Throwable th) {
    System.out.println("Failed from " +
        "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}

// Call the asynchronous command client to run the command.
asyncCmdClientHelper.execute(cmd);
System.out.println("\nCompleted running of the command");

// Check the command result.
CommandResult result = cmd.getCommandResult();
if (result != null) {
    if (result.isSuccessful()) {
        System.out.println("\nCommand ran successfully "
```

```
                                  + "with result\n" + result.getResult());
                }
                else {
                   System.out.println("\nCommand ran with " +
                                            "Exception");
                   result.getException().printStackTrace();
                }
             }
          } catch (Exception e) {
             e.printStackTrace();
          }
      }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

## What to do next

You can complete other tasks associated with business-level applications, such as creating other business-level applications, stopping and starting business-level applications, and so on.

# Deleting an asset using programming

You can delete an asset from a business-level application using programming if the asset is not functioning correctly, the asset is no longer needed, and so on. An asset represents at least one binary file that implements business logic.

## Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interface documentation.

Before you can delete an asset, you must have imported the asset. All the composition units associated with the asset must be deleted using the deleteCompUnit command before you delete the asset. Otherwise, you have to force the deletion. If you do not force the deletion, the deletion fails. If any other composition units have a dependency on a composition unit being deleted with the force option, the deletion fails. After all dependencies on the composition unit are removed, the force option succeeds.

## About this task

You can delete an asset using programming, the administrative console, or the wsadmin tool. Use this topic to delete an asset using programming.

You must specify the assetID parameter of the asset that you are deleting. You might delete an asset if it is not functioning correctly, it is no longer needed, and so on.

Perform the following tasks to delete an asset using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that deletes an asset.

   The command name is deleteAsset. The assetID parameter is a required parameter to specify the asset to delete. You can optionally specify the delete parameter to force deletion of an asset if composition units are still associated with the asset.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to run the command that deletes an asset.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the asset is deleted.

## Example

The following example shows how to delete an asset from a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;
```

```
public class DeleteAsset {

    public static void main(String [] args) {

        try {


            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.

            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                              AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

            // Comment out the previous lines to create a client command
            // manager if you are using a local command manager.
            // Uncomment the following line to create a local command
            // manager.
            //
            // CommandMgr cmdMgr = CommandMgr.getCommandMgr();
            System.out.println("\nCreated command manager");

            // Optionally create an asynchronous command handler.
            // Comment out the following line if no further handling
            // of command notification is required.
            AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

            // Create an asynchronous command client.

            // Set up the session.
            String id = Long.toHexString(System.currentTimeMillis());
            String user = "content" + id;
            Session session = new Session(user, true);

            // If no command handler is used, replace listener with
            // null for the following AsyncCommandClient object:
            AsyncCommandClient asyncCmdClientHelper = new
            AsyncCommandClient(session, listener);
            System.out.println("\nCreated async command client");

            // Create the command that deletes the asset.
            String cmdName = "deleteAsset";
            AdminCommand cmd = cmdMgr.createCommand(cmdName);
            cmd.setConfigSession(session); // Delete the asset from the
                    // business-level application using the session created.
            System.out.println("\nCreated " + cmdName);

            // Set the assetID  parameter to the asset that is to be
            // deleted.
            // Examples of valid formats for the assetID parameter are:
```

```
            // - aName
            // - assetname=aName
            // - WebSphere:assetname=aName
            // This parameter will accept an incomplete ID as long as
            // the incomplete ID can resolve to a unique asset
            // in the business-level application.
            String assetID  = "as1";
            cmd.setParameter("assetID ", assetID );

            System.out.println("\nSet assetID  parameter to "
                               + cmd.getParameter("assetID "));
            // Uncomment the following line of code to set the force parameter
            // to force the deletion even if there are composition units
            // associated with this asset.
            //
            // cmd.setParameter("force", "true");

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                           "parameters");
            } catch (Throwable th) {
                System.out.println("Failed from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Call the asynchronous command client to run the command.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running the command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                  + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                         "Exception");
                    result.getException().printStackTrace();
                }
           }
       } catch (Exception e) {
           e.printStackTrace();
       }
    }
}


package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

### What to do next

You can complete other steps associated with assets in business-level applications, such as adding or deleting other assets, listing assets, exporting assets, and so on.

## Deleting a composition unit using programming

You can delete a composition unit from a business-level application if the composition unit is not functioning correctly, the composition unit is no longer needed, and so on. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can delete a composition unit, you must have created an empty business-level application, imported an asset, and added a composition unit to the business-level application. If other composition units depend on the composition unit that you are deleting and you do not use the force option, the deletion fails.

### About this task

You can delete a composition unit using programming, the administrative console, or the wsadmin tool. This topic describes how to delete a composition unit using programming.

You must provide the blaID and cuID parameters to specify the composition unit that you are deleting from the business-level application.

Perform the following tasks to delete a composition unit using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that deletes a composition unit.

   The command name is deleteCompUnit. The blaID and cuID parameters are required parameters. The cuID parameter is used to specify the composition unit to delete from the business-level application, which is specified with the blaID. You can optionally provide the force parameter to force the deletion if other composition units depend on this composition unit.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to run the command that deletes a composition unit.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the composition unit is deleted.

## Example

The following example shows how to delete a composition unit from a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class DeleteCompUnit {

    public static void main(String [] args) {

        try {


            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);
            // to get the soapClient soap client if
            // you use the local command manager.


            String host = "localhost";
            String port = "8880";   // Change to your port number if it is
                                    // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                              AdminClientFactory.createAdminClient(config);
```

```
// Create the command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace listener with
// null for the following AsyncCommandClient object:
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that deletes the composition unit.
String cmdName = "deleteCompUnit";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Delete the composition unit from
                               // the business-level application
                               // using the session created.
System.out.println("\nCreated " + cmdName);

// Set the blaID parameter to the business-level application with
// the composition unit to delete.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as
// the incomplete ID can resolve to a unique
// business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                   + cmd.getParameter("blaID"));

// Set the cuID parameter to the composition unit that is to be
// deleted.
// Examples of valid formats for the cuID parameter are:
// - name
// - cuname=name
// - WebSphere:cuname=name
// This parameter accepts an incomplete ID as long as the
// incomplete ID can resolve to a unique composition unit
// within the business-level application.
String cuID = "cu1";
cmd.setParameter("cuID", cuID);

System.out.println("\nSet cuID parameter to "
                   + cmd.getParameter("cuID"));
```

```
            // Uncomment the following line of code to set the force parameter
            // to force the deletion even if other composition units depend
            // on this composition unit.
            //
            // cmd.setParameter("force", "true");


            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                           "parameters");
            } catch (Throwable th) {
                System.out.println("Failed from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Call the asynchronous command client to run the command.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of the command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                 }
                 else {
                    System.out.println("\nCommand ran with " +
                                             "Exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

### What to do next

You can complete other tasks associated with the business-level application, such as adding or deleting other composition units, listing composition units, and so on.

## Exporting an asset using programming

You can export an asset from the current session so that you can back up the asset, import the asset to another session, and so on. An asset represents at least one binary file that implements business logic.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

This task assumes that you have already imported an asset.

### About this task

You can export an asset using programming, the administrative console, or the wsadmin tool. This topic describes how to export an asset using programming.

You must provide an assetID parameter value and a file name parameter value to export an asset. The assetID parameter identifies the asset you want to export. An asset ID can take a number of forms. The list below shows various forms for an asset named asset1.jar.

- asset1.jar
- assetname=asset1.jar
- WebSphere:assetname=asset1.jar

The filename parameter specifies a file system file name and location for the exported asset. Specify a fully qualified file path for the file name parameter because the results with relative path names are unpredictable. If you specify a file name parameter of a file that already exists, the file is overwritten with the exported asset.

Perform the following tasks to export an asset from a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that exports an asset.

   The command name is exportAsset. The assetID and filename parameters are required parameters to specify the asset to export and the file name and directory where the asset is exported.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to run the command that exports an asset.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the asset is exported.

## Example

The following example shows how to export an asset from a business-level application based on the previous steps.

Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ExportAsset {

    public static void main(String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.

            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                            AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);
```

```
            // Comment out the previous lines to create a client command
            // manager if you are using a local command manager.
            // Uncomment the following line to create a local command
            // manager:
            //
            // CommandMgr cmdMgr = CommandMgr.getCommandMgr();
            System.out.println("\nCreated command manager");

            // Optionally create an asynchronous command handler.
            // Comment out the following line if no further handling
            // of command notification is required:
            AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

            // Create an asynchronous command client.

            // Set up the session.
            String id = Long.toHexString(System.currentTimeMillis());
            String user = "content" + id;
            Session session = new Session(user, true);

            // If no command handler is used, replace the following listener with
            // null for the AsyncCommandClient object.
            AsyncCommandClient asyncCmdClientHelper = new
            AsyncCommandClient(session, listener);
            System.out.println("\nCreated async command client");

            // Create command that exports the asset.
            String cmdName = "exportAsset";
            AdminCommand cmd = cmdMgr.createCommand(cmdName);
            cmd.setConfigSession(session); // Export as asset
                                     // using the session created.
            System.out.println("\nCreated " + cmdName);

            // (required) Set the assetID parameter to the composition
            // unit that you are exporting.
            // Examples of valid formats for the assetID parameter are:
            // - aName
            // - assetname=aName
            // - WebSphere:assetname=aName
            // This parameter accepts an incomplete ID as long as
            // the incomplete ID can resolve to a unique asset
            // within the business-level application.
            String assetID = "test5.zip";
            cmd.setParameter("assetID", assetID);

            System.out.println("\nSet assetID parameter to "
                                + cmd.getParameter("assetID"));

            // Set the file name for the asset to be exported. Use a
            // fully qualified path name.  An existing file with the specified
            // name will be overwritten.
DownloadFile filename = new DownloadFile("/assets/asset1.zip");


            cmd.setParameter("filename", filename);

            System.out.println("\nSet filename parameter to "
                                + cmd.getParameter("filename"));

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                           "parameters");
            } catch (Throwable th) {
                 System.out.println("Failed from " +
```

```
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Call the asynchronous command client to run the command.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of the command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                 }
                 else {
                    System.out.println("\nCommand ran with " +
                                             "Exception");
                    result.getException().printStackTrace();
                    }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

You can import the asset to another session. You can complete other tasks associated with assets, such as listing assets, and editing assets.

## Listing assets using programming

You can list the assets that have been imported so that you can do further asset administration, such as deleting or exporting assets. An asset represents at least one binary file that implements business logic.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

You can list assets using programming, the administrative console, or the wsadmin tool.

### About this task

You can list assets using programming, the administrative console, or the wsadmin tool. This topic describes how to list assets using programming.

When you list assets, all the assets are listed unless you set the assetID to specify the asset that you want to list. You can optionally include deployable units or a description of the assets when you list the assets. After you list the assets, you can use the information to do further administration, such as deleting or exporting assets.

Perform the following tasks to list assets using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that lists assets.

   The command name is listAssets. You can optionally set the assetID parameter to query for assets that match the ID. You can also optionally set the includeDescription parameter and the includeDeplUnit parameter to include the display of the asset description and its deployable units.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to list the asset.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, a list of assets is displayed.

## Example

The following example shows how to list the assets based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
```

```
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ListAssets {

    public static void main(String[] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.


            String host = "localhost";
            String port = "8880"; // Change to your port number if
                                  // it is not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                      AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

            // Comment out the previous lines to create a client command
            // manager if you are using a local command manager.
            // Uncomment the following line to create a local command
            // manager:
            //
            // CommandMgr cmdMgr = CommandMgr.getCommandMgr();
            System.out.println("\nCreated command manager");

            // Optionally create an asynchronous command handler.
            // Comment out the following line if no further handling
            // of command notification is required.
            AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

            // Create an asynchronous command client.

            // Set up the session.
            String id = Long.toHexString(System.currentTimeMillis());
            String user = "content" + id;
            Session session = new Session(user, true);

            // If no command handler is used, replace listener with
            // null for the AsyncCommandClient object.
            AsyncCommandClient asyncCmdClientHelper = new
                       AsyncCommandClient(session, listener);
            System.out.println("\nCreated async command client");

            // Create the command that lists the assets.
            String cmdName = "listAssets";
            AdminCommand cmd = cmdMgr.createCommand(cmdName);
            cmd.setConfigSession(session); // list all the assets
                                  // using the session created.
```

```
        System.out.println("\nCreated " + cmdName);

        // Optionally set the assetID parameter.
        // Uncomment the following code to set the assetID parameter to
        // only list the asset with the ID specified, otherwise all
        // assets are listed. Change the assetID parameter according to your
        // scenario.
        // Examples of valid formats for the assetID parameter are:
        // - aName
        // - assetname=aName
        // - WebSphere:assetname=aName
        // All assets that match the ID specification are listed.
        // The ID must include at least the asset name.
        // String assetID = "asset1.zip";
        // cmd.setParameter("assetID", assetID);

        //System.out.println("\nSet assetID parameter to "
        //                     + cmd.getParameter("assetID"));

        // Optionally include a description by setting
        // the includeDescription parameter to true or false.
        String includeDescription = "true";
        cmd.setParameter("includeDescription", includeDescription);

        System.out.println("\nSet includeDescription parameter to "
                           + cmd.getParameter("includeDescription"));

        // Optionally include deployable units by setting
        // the includeDeplUnit parameter to true or false.
        String includeDeplUnit = "false";
        cmd.setParameter("includeDeplUnit", includeDeplUnit);

        System.out.println("\nSet includeDeplUnit parameter to "
                            + cmd.getParameter("includeDeplUnit"));

        // Call the asynchronous client helper to process parameters.
        try {
            asyncCmdClientHelper.processCommandParameters(cmd);
            System.out.println("\nCompleted process command " +
                                  "parameters");
        } catch (Throwable th) {
            System.out.println("Failed from " +
                "asyncCmdClientHelper.processCommandParameters(cmd).");
            th.printStackTrace();
            System.exit(-1);
        }

        // Run the command to list assets.
        asyncCmdClientHelper.execute(cmd);
        System.out.println("\nCompleted running of the command");

        // Check the command result.
        CommandResult result = cmd.getCommandResult();
        if (result != null) {
            if (result.isSuccessful()) {
                System.out.println("\nCommand ran successfully "
                                     + "with result\n" + result.getResult());
            }
            else {
                System.out.println("\nCommand ran with " +
                                              "Exception");
                result.getException().printStackTrace();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
    '}
}


package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

You can complete other tasks associated with assets, such as deleting, editing, and exporting assets.

## Listing composition units using programming

You can list the composition units for a specific business-level application so that you can complete further composition unit administration, such as deleting or adding composition units. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can list composition units, you must have imported an asset, created an empty business-level application, and added a composition unit to the business-level application.

### About this task

You can list composition units using programming, the administrative console, or the wsadmin tool. This topic describes how to list composition units using programming.

You must provide the blaID parameter to specify the business-level application to list the composition unit. When you list composition units for a business-level application, you can optionally list the type for each composition unit and the description for each composition unit that has a description. You can use the list to complete further administration, such as deleting or exporting composition units.

Perform the following tasks to list composition units for a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.

4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Use the command manager that you created in a previous step to create and set up the command that lists composition units.

   The command name is listCompUnits. The blaID parameter is a required parameter that you use to specify the business-level application to list the composition units. You can optionally set the includeDescription parameter to display the composition unit descriptions. You can also optionally set the includeType parameter to display the composition unit types.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to list the composition units.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, a list of composition units for a business-level application is displayed.

## Example

The following example shows how to list the composition units of a specific business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ListCompUnits {

    public static void main(String[] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
```

```
// to get the soapClient soap client if you use the local
// command manager.

String host = "localhost";
String port = "8880"; // Change to your port number if
                              // it is not 8880.

Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
                   AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
            AdminClientFactory.createAdminClient(config);

// Create the command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
              AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that lists the composition units.
String cmdName = "listCompUnits";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // List all the compostion units
                               // for the business-level application
                               // with this session ID.
System.out.println("\nCreated " + cmdName);

// Set the blaID parameter to the business-level application
// whose composition units are listing.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                     + cmd.getParameter("blaID"));
```

```
            // Optionally include descriptions for each composition unit
            // that has a description by setting
            // the includeDescription parameter to true or false.
            String includeDescription = "true";
            cmd.setParameter("includeDescription", includeDescription);

            System.out.println("\nSet includeDescription parameter to "
                                + cmd.getParameter("includeDescription"));
            // Optionally include types for each composition unit
            // by setting the includeType parameter to true or false.
            String includeType = "true";
            cmd.setParameter("includeType", includeType);

            System.out.println("\nSet includeType parameter to "
                            + cmd.getParameter("includeType"));

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                            "parameters");
            } catch (Throwable th) {
                System.out.println("Failed from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Run the command to list the composition units.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                         "Exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

### What to do next

Now that you have listed the composition units for a business-level application, you can complete other tasks associated with composition units, such as adding or deleting composition units.

## Listing business-level applications using programming

You can list the business-level applications of a session so that you can complete further business-level application administration such as deleting a business-level application. A business-level application is an administrative model that captures the definition of an enterprise-level application so that you can perform specific business functions, such as accounting.

### Before you begin

Before you can list business-level applications of a session, you must have created an empty business-level application.

### About this task

You can list business-level applications of a session using programming, the administrative console, or the wsadmin tool. This topic describes how to list business-level applications using programming.

List all the business-level applications of a session unless you set the blatID parameter to specify the business-level application that you want to list. You can optionally list the business-level applications with a description for those that have a description if you set the includeDescription parameter to `true`. After you list the business-level applications, you can use the information to do further administration, such as starting or deleting business-level applications.

Perform the following tasks to list business-level applications of a session using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that lists business-level applications of a session.

   The command name is listBLAs. You can optionally set the blaID parameter to query for business-level applications that match the ID. You can optionally set the includeDescription parameter to display the business-level application descriptions.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to list the business-level applications of a session.

You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, a list of business-level applications for a session is displayed.

## Example

The following example shows how to list the business-level applications of a session based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ListBLAs {

        public static void main(String[] args) {

            try {

                // Connect to the application server.
                // This step is optional if you use the local
                // command manager. Comment out the lines to and including
                // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
                // soapClient);
                // to get the soapClient soap client if you use the local
                // command manager.

                String host = "localhost";
                String port = "8880"; // Change to your port number if
                                              // it is not 8880.

                Properties config = new Properties();
                config.put(AdminClient.CONNECTOR_HOST, host);
                config.put(AdminClient.CONNECTOR_PORT, port);
                config.put(AdminClient.CONNECTOR_TYPE,
                                   AdminClient.CONNECTOR_TYPE_SOAP);
                System.out.println("Config: " + config);
                AdminClient soapClient =
                                  AdminClientFactory.createAdminClient(config);

                // Create the command manager.
                CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

                // Comment out the previous lines to create a client command
                // manager if you are using a local command manager.
                // Uncomment the following line to create a local command
                // manager:
                //
```

```
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace listener with
// null for the AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that lists the business-level applications.
String cmdName = "listBLAs";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // list all the business-level applications
                              // using the session created.

System.out.println("\nCreated " + cmdName);

// Optionally set the blaID parameter.
// Uncomment the following code to set the blaID parameter to
// only list the business-level applications with the ID specified. Otherwise all
// business-level applications are listed. Change the blaID parameter according
// to your scenario.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// All business-level applications that match the ID specification
// are listed. The ID must include at least the business-level
// application name.
// String blaID = "bla1";
// cmd.setParameter("blaID", blaID);

//System.out.println("\nSet blaID parameter to "
//                    + cmd.getParameter("blaID"));

// Optionally include a description by setting
// the includeDescription parameter to true instead of false.
String includeDescription = "true";
cmd.setParameter("includeDescription", includeDescription);

System.out.println("\nSet includeDescription parameter to "
                    + cmd.getParameter("includeDescription"));


// Call the asynchronous client helper to process parameters.
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
    System.out.println("\nCompleted process command " +
                       "parameters");
} catch (Throwable th) {
    System.out.println("Failed from " +
        "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}
```

```
            // Run the command to list business-level applications.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of the command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                     System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                            "Exception");
                   result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

### What to do next

You can complete other tasks associated with business-level applications, such as deleting, starting, or stopping business-level applications.

## Editing a composition unit using programming

You can edit the configuration information in a composition unit of a business-level application if, for example, you want to change certain modules in the composition unit that are configured to run in specific targets. A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can edit a composition unit of a business-level application, you must have created an empty business-level application, imported an asset, and added a composition unit to the business-level application.

## About this task

You can edit a composition unit of a business-level application using programming, the administrative console, or the wsadmin tool. This topic describes how to edit a composition unit of a business-level application using programming.

You must provide the blaID and cuID parameters to specify the composition unit of the business-level application that you are editing.

Perform the following tasks to edit a composition unit of a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command that edits a composition unit of a business-level application.

   The command name is editCompUnit. Use the required blaID and cuID parameters to specify the composition unit of the business-level application that you are editing.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Set up the command step parameters.

   You can edit various composition unit information through steps. The CUOptions step contains data about the composition unit such as its description, starting weight, and start and restart behavior. The MapTargets step contains target information about where to deploy the composition unit. The RelationshipOptions step contains shared library composition units on which this composition unit has a dependency. The ActivationPlanOptions step allows you to change runtime components for each deployable unit. You can edit parameters in these steps.
8. Call the asynchronous command client to run the command that edits a composition unit of a business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
9. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the composition unit of a business-level application is edited.

## Example

The following example shows how to edit a composition unit of a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class EditCompUnit {

    public static void main(String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.

            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                              AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

           // Comment out the previous lines to create a client command
           // manager if you are using a local command manager.
           // Uncomment the following line to create a local command
           // manager:
           // CommandMgr cmdMgr = CommandMgr.getCommandMgr();
           System.out.println("\nCreated command manager");

           // Optionally create an asynchronoous command handler.
           // Comment out the following line if no further handling
           // of command notification is required:
           AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

           // Create an asynchronous command client.
```

```
// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the following AsyncCommandClient object:
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that edits the composition unit.
String cmdName = "editCompUnit";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Edit a certain composition
       // unit of a business-level using the session created.
  System.out.println("\nCreated " + cmdName);

// Set the blaID parameter.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as the
// incomplete ID can resolve to a unique business-level application.
String blaID = "bla1";
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
                    + cmd.getParameter("blaID"));

// Set the cuID parameter.
// Examples of valid formats for the cuID parameter are:
// - name
// - cuname=name
// - WebSphere:cuname=name
// This parameter accepts an incomplete ID as long as the
// incomplete ID can resolve to a unique composition unit
// within the business-level application.
String cuID = "cu1";
cmd.setParameter("cuID", cuID);

System.out.println("\nSet cuID parameter to "
                    + cmd.getParameter("cuID"));

// Call the asynchronous client helper to process the command parameters.
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
    System.out.println("\nCompleted process command " +
                        "parameters");
} catch (Throwable th) {
    System.out.println("Failed from " +
        "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}

// Set up the step parameters for the CUOptions step.
// The CUOptions step contains the following arguments that can be edited:
// description - description for the composition unit
// startingWeight - starting weight for the composition unit
//                  within the business-level application.
// startedOnDistributed - to start composition unit upon distribution
//                        to target nodes.
// Valid values are true, false.
// restartBehaviorOnUpdate - restart behavior for the composition
```

```
//                  unit when the compostion unit is updated.
// Valid values are DEFAULT, ALL, NONE
String stepName = "CUOptions";
CommandStep step = ((TaskCommand) cmd).gotoStep(stepName);

// Composition Unit description:
String description = "cu1 description changed in editCompUnit";

for(int i = 0; i < step.getNumberOfRows(); i++) {
     // Use the following code to change the composition unit step parameters
     // of the CUOptions step. Change your set of step parameters
     // as required by your scenario.

     // For example, set the description.
     step.setParameter("description", description, i);
     System.out.println("\nSet description parameter to " +
                        step.getParameter("description", i));
}

// Set up the step parameters for the MapTargets step
stepName = "MapTargets";
step = ((TaskCommand) cmd).gotoStep(stepName);

// In this step the server parameter is required.
// server - target(s) to deploy the composition unit. The default is server1.
//          To add an additional target to the existing
//          target, add a prefix to the target with a "+". To
//          delete an existing target, add a prefix to the
//          target with a "#". To replace the existing
//          target, use the regular syntax as in the addCompUnit example.
// Example: server = "#server1+server2";
          String server = "server1";

for(int i = 0; i < step.getNumberOfRows(); i++) {
     // Use the following code to set the server parameter of the MapTargets step.
      // Change your set of step parameters as required by your
      // scenario.

     // For example, set the server.
     step.setParameter("server", server, i);
     System.out.println("\nSet server parameter to " +
                        step.getParameter("server", i));
}

// If the RelationshipOptions step is available, the selected
// deployable units of the source asset of the "primary" composition
// unit (that is, the composition unit being added) have dependencies
// on other assets for which there are matching "secondary" composition
// units in the business-level application.  The RelationshipOptions step is much like
// CreateAuxCUOptions except that the required secondary composition
// units already exist.  Also, each RelationshipOptions row maps one
// deployable unit to one or more secondary composition units, whereas,
// each CreateAuxCUOptions row maps one deployable unit to one
// asset dependency.
//
// Each RelationshipOptions row corresponds to one deployable unit
// with one or more dependency relationships and consists of
// parameter values for the dependency relationships.  Some parameters
// are read-only and some of them are editable.  To edit parameter
// values, use the same approach as that used to edit parameter values
// in the CUOptions step.
//
// The parameters for this step include:
//
//  deplUnit – The name of the deployable unit which has the
//             dependency. (Read-only.)
//  relationship – Composition unit dependencies in the form of a
```

```
    //                 list of composition unit IDs.  Composition unit
    //                 IDs are separated by a "plus" sign ("+").  Each ID
    //                 can be fully or partially formed as shown with the
    //                 following examples:
    //                     WebSphere:cuname=SharedLib1.jar
    //                     WebSphere:cuname=SharedLib.jar
    //                     SharedLib.jar
    //  matchTarget - Specifies whether the server target for the secondary
    //                composition units are to match the server target for
    //                the primary composition unit.  The default value
    //                is "true".  If the value is set to "false", the
    //                secondary composition unit will be created with no
    //                target.  The target on the secondary composition unit
    //                can be set at a later time with the editCompUnit
    //                command.
    // for(int i = 0; i < step.getNumberOfRows(); i++) {
         // Use the following if statement to set the relationship and matchTarget parameters
         // of the RelationshipOptions step. Change your set of
         // step parameters as required by your scenario.

         // Uncomment the following code to match the deplUnit and then set
         // the relationship differently.
         //String deplUnit = (String) step.getParameter("deplUnit",
         //                                                       i);

         //if (deplUnit.equals("a1.jar") {
             // For example, change the relationship for the a1.jar file.
             //step.setParameter("relationship", relationship, i);
             //System.out.println("\nSet relationship parameter " +
             //       "to " + step.getParameter("relationship", i));

             // For example, change matchTarget.
             //step.setParameter("matchTarget", matchTarget, i);
             //System.out.println("\nSet matchTarget parameter to "+
             //             step.getParameter("matchTarget", i));
         //}
    //}

    // The addCompUnit command contains thr ActivationPlanOptions step.
    // The user can set the ActivationPlanOptions step parameters similar to
    // the step parameters for the CUOptions step in the previous examples.
    // The arguments for this step include:
    // deplUnit – deployable unit URI (read only parameter)
    // activationPlan - specifies a list of runtime components in the
    // format of specname=xxxx

    // Run the command command to edit the composition unit.
    asyncCmdClientHelper.execute(cmd);
    System.out.println("\nCompleted running of the command");

    // Check the command result.
    CommandResult result = cmd.getCommandResult();
    if (result != null) {
        if (result.isSuccessful()) {
            System.out.println("\nCommand ran successfully "
                           + "with result\n" + result.getResult());
        }
         else {
            System.out.println("\nCommand ran with " +
                                 "Exception");
            result.getException().printStackTrace();
            }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

```
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

After you edit the composition unit, you can run the updated business-level application.

## Editing an asset using programming

You can edit the information of an asset such as its destination location, its relationship with other assets, and so on. An asset represents at least one binary file that implements business logic.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can edit an asset, you must have imported an asset.

You can edit an asset of a business-level application using programming, the administrative console, or the wsadmin tool.

### About this task

You can edit an asset of a business-level application using programming, the administrative console, or the wsadmin tool. This topic describes how to edit an asset of a business-level application using programming.

You must provide the assetID parameter to specify the asset that you are editing.

Perform the following tasks to edit an asset of a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Use the command manager that you created in a previous step to create and set up the command that edits an asset.

   The command name is editAsset. The assetID parameter is a required parameter to specify the asset that you are editing.
6. Call the asynchronous command client to process the command parameters.
7. Set up the command step parameters.

   The AssetOptions step contains data about the asset such as its description, file permission, and relationship with other assets. You can edit various parameters in the AssetOptions step.
8. Call the asynchronous command client to run the command that edits an asset of a business-level application.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
9. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the asset of a business-level application is edited.

## Example

The following example shows how to edit an asset of a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class EditAsset {

    public static void main(String [] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager.

            // Comment out the lines to and including get the
            // soapClient soap client if you use the local command manager.
            // Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
```

```
// soapClient);
// to get the soapClient soap client if you use the
// local command manager.
String host = "localhost";
String port = "8880"; // Change to your port number if it is
                      // not 8880.

Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
           AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
                 AdminClientFactory.createAdminClient(config);

// Create the command manager:
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager.
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create async command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the following AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that edits the asset.
String cmdName = "editAsset";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Edit an imported asset
                               // using the session created.
System.out.println("\nCreated " + cmdName);

// Set the assetID parameter
// Examples of valid formats for the assetID parameter are:
// - aName
// - assetname=aName
// - WebSphere:assetname=aName
// This parameter accepts an incomplete ID as long as
// the incomplete ID can resolve to a unique asset.
String assetID = "asset1.zip";
cmd.setParameter("assetID", assetID);

System.out.println("\nSet assetID parameter to "
                   + cmd.getParameter("assetID"));

// Call the asynchronous client helper to process parameters.
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
```

```
    System.out.println("\nCompleted process command " +
                            "parameters");
} catch (Throwable th) {
    System.out.println("Failed from " +
        "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}

// Set up the step parameters for the AssetOptions step.
String stepName = "AssetOptions";
CommandStep step = ((TaskCommand) cmd).gotoStep(stepName);

// Asset description:
String description = "asset for testing";

// destination of deployed asset
String destinationUrl = "/myInstalledAssets/asset1.zip";
// Asset type aspect:
String typeAspect = "spec=sharedlib";

// Asset validation:
String validate = "yes";

// Permission of files:
String filePermission = ".*\\.dll=755";

// Asset relationship:
String relationship = "";

for (int i = 0; i < step.getNumberOfRows(); i++) {
    // The following lines set the description and typeAspect
    // step parameters. There are other step parameters
    // in the AssetOptions step in the following comments. Change your set
    // of step parameters as required by your scenario.

    // For example, set description
    step.setParameter("description", description, i);
    System.out.println("\nSet description parameter to " +
          step.getParameter("description", i));

    // For example, set the typeAspect parameter.
    // Format of a typeAspect is:
    // WebSphere:spec=xxx,version=n.n+
    // WebSphere:spec=xxx,version=n.n
    step.setParameter("typeAspect", typeAspect, i);
    System.out.println("\nGet typeAspect: " +
          step.getParameter("typeAspect", i));

    // For example, set the destination parameter.
    step.setParameter("destination", destination, i);
    System.out.println("\nSet destination parameter to " +
             step.getParameter("destination", i));

    // For example, set the validate parameter.
    step.setParameter("validate", validate, i);
    System.out.println("\nSet validate parameter to " +
            step.getParameter("validate", i));

    // For example, set the filePermission perameter.
    step.setParameter("filePermission", filePermission, i);
    System.out.println("\nSet filePermission parameter to " +
          step.getParameter("filePermission", i));

    // For example, set relationship.
    step.setParameter("relationship", relationship, i);
```

```
                System.out.println("\nSet relationship paramter to " +
                        step.getParameter("relationship", i));
            }

            // Run the command to edit the asset.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                         "Exception");
                    result.getException().printStackTrace();
                    }
            }
         } catch (Exception e) {
             e.printStackTrace();
         }
    }
}
```

```
package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

## What to do next

After you edit the asset, you can add the asset as a composition unit to a business-level application, or export the asset.

# Editing a business-level application using programming

You can edit the information of a business-level application such as its description. A business-level application is an administrative model that captures the entire definition of an enterprise-level application.

## Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can edit a business-level application, you must have created a business-level application.

You can edit a business-level application using programming, the administrative console, or the wsadmin tool.

## About this task

You must provide the blaID parameter to specify the business-level application that you are editing.

Perform the following tasks to edit a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create the command that edits a business-level application.

   The command name is editBLA. Use the required blaID parameter to specify the business-level application that you are editing.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Set up the command step parameter by setting the description parameter.

   The BLAOptions step contains a description for the business-level application. You can edit the description parameter in the BLAOptions step.
8. Call the asynchronous command client to run the command to edit a business-level application.

   You could have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
9. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the business-level application is edited.

## Example

The following example shows how to edit a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
```

```
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class EditBLA {

    public static void main(String[] args) {
        try {

            // Connect to the application server.
            // This step is optional if you use the local command manager.
            // Comment out the following lines to get the soapClient SOAP client if
            // you are going to use the local command manager. You would
            // comment out the lines to and including
            // CommandMgr cmdMgr =
            // CommandMgr.getClientCommandMgr(soapClient);

            String host = "localhost"; // Change to your host if it is not localhost.
            String port = "8880"; // Change to your port number if it is not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                   AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                             AdminClientFactory.createAdminClient(config);

            // Create command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

            // Comment out the previous lines to create a client command
            // manager if you are using a local command manager.
            // Uncomment the following line to create a local command
            // manager.
            //
            // CommandMgr cmdMgr = CommandMgr.getCommandMgr();

            System.out.println("\nCreated command manager");

            // Optionally create an asynchronous command handler.
            // Comment out the following line if no further handling
            // of command notification is required.
            AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

            // Create an asynchronous command client.

            // Set up the session.
            // This example creates a new session. You can replace the
            // code below to use an existing session that has been
            // created.
            String id = Long.toHexString(System.currentTimeMillis());
            String user = "content" + id;
            Session session = new Session(user, true);

            // If no command handler is used, replace the listener with
            // null for the following AsyncCommandClient object.
            AsyncCommandClient asyncCmdClientHelper = new
                           AsyncCommandClient(session, listener);
            System.out.println("\nCreated async command client");
```

```
// Create the command that edits the business-level application.
String cmdName = "editBLA";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Edit an existing business-level
                               // application using the session
                               // created.
System.out.println("\nCreated " + cmdName);


// Set the blaID parameter (required).
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";   // Replace bla1 with your value of the blaID.
cmd.setParameter("blaID", blaID);

System.out.println("\nSet blaID parameter to "
        + cmd.getParameter("blaID"));

// Call the asynchronous client helper to process parameters.
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
    System.out.println("\nCompleted process command " +
                       "parameters");
} catch (Throwable th) {
    System.out.println("Throwing an exception from " +
            "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}

// Set up the step parameters for the BLAOptions step.
// The only step parameter you can edit is description.
String stepName = "BLAOptions";
CommandStep step = ((TaskCommand) cmd).gotoStep(stepName);

// Edit the business-level application description.
String description = "bla for testing"; // Replace with your value.

for (int i = 0; i < step.getNumberOfRows(); i++) {
    // The following lines set the description
    // step parameter.
    step.setParameter("description", description, i);
    System.out.println("\nSet description parameter to " +
            step.getParameter("description", i));
}

// Run the command to edit the business-level application.
asyncCmdClientHelper.execute(cmd);
System.out.println("\nCompleted command execution");

// Check the command result.
CommandResult result = cmd.getCommandResult();
if (result != null) {
    if (result.isSuccessful()) {
        System.out.println("\nCommand executed successfully "
                + "with result\n" + result.getResult());
    }
    else {
        System.out.println("\nCommand executed with " +
                           "Exception");
        result.getException().printStackTrace();
    }
}
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification.
        System.out.println("\nEXAMPLE: notification received: " +
                            notification);
    }
}
```

### What to do next

After you edit the business-level application, you can continue administration of business-level applications. You can do such things as start and stop a business-level application, delete a business-level application, add a composition unit to a business-level application, and so on.

## Updating an asset using programming

You can update an asset by adding, deleting, or updating a single file or Java Platform, Enterprise Edition (Java EE) module, or by merging multiple files or Java EE modules into an asset. You can also update an asset by replacing the entire asset.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interface documentation.

Before you can update an asset, you must have imported the asset.

You can update an asset using programming, the administrative console, or the wsadmin tool.

### About this task

You must specify the assetID parameter of the asset that you are updating. In addition, you must specify the operation parameter. Wether or not you must specify the contents and contenturi parameters depends on the operation that you specify.

You modify one or more files or module files of an asset with this task. You also update the asset binary file, but do not update the composition units that the system deploys with this asset as a backing object.

Perform the following tasks to update an asset using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.

4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Create and set up the command that updates an asset.
   a. Set the parameter for the asset that you are updating.
   b. Set the operation parameter.
   c. Set the contents parameter unless the operation is set to `delete`.
   d. Set the contenturi parameter if the operation is set to `add`, `update`, or `addupdate`.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to run the command to update an asset.

   You could have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, the asset is updated.

## Example

The following example shows how to update an asset based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class EditBLA {

    public static void main(String[] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local command manager.
            // Comment out the following lines to get soapClient soap client if
```

```
// you are going to use the local command manager.
// Comment out the lines to and including
// CommandMgr cmdMgr =
// CommandMgr.getClientCommandMgr(soapClient);

String host = "localhost"; // Change to your host if it is not localhost.
String port = "8880"; // Change to your port number if it is not 8880.

Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
       AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
                  AdminClientFactory.createAdminClient(config);

// Create the command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager.
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();

System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required.
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
// This example creates a new session. You can replace the
// following code to use an existing session that has been
// created.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If you do not use the command handler, replace the listener with
// null for the following AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
                AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that updates the asset.
String cmdName = "updateAsset";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // Update an asset
                               // using the session
                               // created.
System.out.println("\nCreated " + cmdName);

// Set the required assetID parameter.
// Examples of valid formats for the assetID parameter:
// - aName
// - assetname=aName
// - WebSphere:assetname=aName
// This parameter accepts an incomplete ID as long as the
// incomplete ID can resolve to a unique asset within the
// business-level application.
String assetID = "asset1.zip";   // Replace asset1.zip with your
```

```
                                 // value of the assetID parameter.
            cmd.setParameter("assetID", assetID);

            System.out.println("\nSet assetID parameter to "
                    + cmd.getParameter("assetID"));

            // Set the required operation parameter.
            // Possible operation values are add, addupdate, delete, merge,
            // replace, and update.
            // Use the add value to add a new file or Java EE module to the asset.
            // Use the addupdate value to add a new file or Java EE module to the asset, or
            // update an existing file or Java EE module.
            // Use the delete value to delete an existing file or Java EE module in the asset.
            // Use the merge value to provide a partial update with multiple
            // additions, updates, or deletions.
            // Use the replace value for a full update to replace all the contents.
            // Use the update value to update an existing file or Java EE module in the asset.
            String op = "add"; // Replace the add value with your operation value.
            cmd.setParameter("operation", op);

            System.out.println("\nSet operation parameter to "
                    + cmd.getParameter("operation"));

            // Set the contents parameter.
            // This parameter is required unless the operation is set to
            // delete.
String contents = "/assets/abc.txt"

            cmd.setParameter("contents", contents);

            System.out.println("\nSet contents parameter to "
                    + cmd.getParameter("contents"));

            // Set the contenturi parameter.
            // This parameter is required for the
            //   add, addupdate, update, or delete operations.
            String contenturi = "abc.txt";  // URI within the asset to
                                            // place the new file. Replace
                                            // with your value.
            cmd.setParameter("contenturi", contenturi);

            System.out.println("\nSet contenturi parameter to "
                    + cmd.getParameter("contenturi"));

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                   "parameters");
            } catch (Throwable th) {
                System.out.println("Throwing an exception from " +
                   "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Run the command.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted command execution");

            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand executed successfully "
                           + "with result\n" + result.getResult());
                }
```

```
            else {
                System.out.println("\nCommand executed with " +
                                   "Exception");
                result.getException().printStackTrace();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification.
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

You can do other tasks associated with assets in business-level applications, such as adding or deleting other assets, listing assets, exporting assets, and so on.

## Viewing a composition unit using programming

A composition unit is typically created from a business-level application or an asset and contains configuration information that makes the asset runnable. You can view the composition unit information so that you can complete other tasks associated with the composition unit such as editing an asset or delete a composition unit.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interfaces documentation.

Before you can view a composition unit of a business-level application, you must have created an empty business-level application, imported an asset into the business-level application, and added a composition unit to the business-level application.

### About this task

You can view a composition unit using programming, the administrative console, or the wsadmin tool. This topic describes how to view a composition unit using programming.

You must provide the blaID and cuID parameters to specify the composition unit of the business-level application that you are viewing. You can view configuration information of the composition unit of a business-level application. The configuration information identifies the asset from which the composition unit is created if the composition unit contains an asset. You can also view runtime targets on which the deployable units of the composition unit are to run.

Perform the following tasks to view a composition unit of a business-level application using programming.

1. Connect to the application server.

The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.

2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.

3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.

4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Use the command manager that you created in a previous step to create and set up the command to view a composition unit.

   The command name is viewCompUnit. Use the required blaID and cuID parameters to specify the composition unit of the business-level application that you are viewing.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Call the asynchronous command client to run the command and view a composition unit.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, you can view the configuration information of a composition unit for a business-level application.

## Example

The following example shows how to view a composition unit of a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ViewCompUnit {
```

```
public static void main(String [] args) {

    try {
        // Connect to the application server.
        // This step is optional if you use the local
        // command manager. Comment out the lines to and including
        // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
        // soapClient);
        // to get the soapClient soap client if you use the local
        // command manager.
        String host = "localhost";
        String port = "8880"; // Change to your port number if it is
                              // not 8880.

        Properties config = new Properties();
        config.put(AdminClient.CONNECTOR_HOST, host);
        config.put(AdminClient.CONNECTOR_PORT, port);
        config.put(AdminClient.CONNECTOR_TYPE,
                   AdminClient.CONNECTOR_TYPE_SOAP);
        System.out.println("Config: " + config);
        AdminClient soapClient =
                          AdminClientFactory.createAdminClient(config);

        // Create the command manager.
        CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

        // Comment out the previous lines to create a client command
        // manager if you are using a local command manager.
        // Uncomment the following line to create a local command
        // manager:
        //
        // CommandMgr cmdMgr = CommandMgr.getCommandMgr();
        System.out.println("\nCreated command manager");

        // Optionally create an asynchronous command handler.
        // Comment out the following line if no further handling
        // of command notification is required:
        AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

        // Create an asynchronous command client.

        // Set up the session.
        String id = Long.toHexString(System.currentTimeMillis());
        String user = "content" + id;
        Session session = new Session(user, true);

        // If no command handler is used, replace the following listener with
        // null for the AsyncCommandClient object.
        AsyncCommandClient asyncCmdClientHelper = new
        AsyncCommandClient(session, listener);
        System.out.println("\nCreated async command client");

        // Create the command that views the composition unit.
        String cmdName = "viewCompUnit";
        AdminCommand cmd = cmdMgr.createCommand(cmdName);
        cmd.setConfigSession(session); // View a certain composition
                                       // unit of a business-level
                                       // application using the session created.
        System.out.println("\nCreated " + cmdName);

        // (required) Set the blaID parameter.
        // Examples of valid formats for the blaID parameter are:
        // - bName
        // - blaname=bName
        // - WebSphere:blaname=bName
        // This parameter accepts an incomplete ID as long as the incomplete
        // ID can resolve to a unique business-level application.
```

```
            String blaID = "bla1";
            cmd.setParameter("blaID", blaID);

            System.out.println("\nSet blaID parameter to "
                               + cmd.getParameter("blaID"));

            // (required) Set the cuID parameter to the composition unit.
            // The cuID parameter has the format of
            // WebSphere:cuname=name. This parameter
            // accepts an incomplete ID as long as the incomplete
            // ID can resolve to a unique composition unit within the
            // business-level application.
            String cuID = "cu1";
            cmd.setParameter("cuID", cuID);

            System.out.println("\nSet cuID parameter to "
                               + cmd.getParameter("cuID"));

            // Call the asynchronous client helper to process parameters
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                     "parameters");
            } catch (Throwable th) {
                System.out.println("Failed from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Call the asynchronous command client to run the command.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of the command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                  + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                           "Exception");
                    result.getException().printStackTrace();
                  }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {
    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

## What to do next

You can use the information that you viewed about the composition unit to perform other tasks. For instance, you might edit the asset in the composition unit to make improvements to the asset. You might export the composition unit, and then import that composition unit into another business-level application.

# Viewing an asset using programming

You can view the asset information so that you can complete other tasks associated with the asset, such as editing or exporting an asset. An asset represents at least one binary file that implements business logic.

## Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interface documentation.

Before you can view an asset of a business-level application, you must have imported an asset.

## About this task

You can view an asset using programming, the administrative console, or the wsadmin tool. This topic describes how to view an asset using programming.

You must provide the assetID parameter to specify the asset you are viewing. You can view configuration information of an asset, such as the destination location and relationships with other assets.

Perform the following tasks to view an asset of a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.
5. Use the command manager that you created in a previous step to create and set up the command to view an asset.

   The command name is viewAsset. Use the required assetID parameter to specify the asset that you are viewing.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to run the command and view an asset.

You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, you can view the configuration information of an asset.

## Example

The following example shows how to view an asset based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class ViewAsset {

    public static void main(String [] args) {

        try {
            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.
            String host = "localhost";
            String port = "8880"; // Change to your port number if it is
                                  // not 8880.

            Properties config = new Properties();
            config.put(AdminClient.CONNECTOR_HOST, host);
            config.put(AdminClient.CONNECTOR_PORT, port);
            config.put(AdminClient.CONNECTOR_TYPE,
                       AdminClient.CONNECTOR_TYPE_SOAP);
            System.out.println("Config: " + config);
            AdminClient soapClient =
                             AdminClientFactory.createAdminClient(config);

            // Create the command manager.
            CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

            // Comment out the previous lines to create a client command
            // manager if you are using a local command manager.
            // Uncomment the following line to create a local command
            // manager:
            //
```

```
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the following listener with
// null for the AsyncCommandClient object:
AsyncCommandClient asyncCmdClientHelper = new
AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command to view the asset.
String cmdName = "viewAsset";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // View a certain composition
                               // unit of a business-level application
                               // using the session created.
System.out.println("\nCreated " + cmdName);

// (required) Set the assetID parameter to the asset.
// Examples of valid formats for the assetID parameter:
// - aName
// - assetname=aName
// - WebSphere:assetname=aName
// This parameter accepts an incomplete ID as long as the
// incomplete ID can resolve to a unique asset.
String assetID = "asset1.zip";
cmd.setParameter("assetID", assetID);

System.out.println("\nSet assetID parameter to "
                    + cmd.getParameter("assetID"));

// Call the asynchronous client helper to process parameters
try {
    asyncCmdClientHelper.processCommandParameters(cmd);
    System.out.println("\nCompleted process command " +
                            "parameters");
} catch (Throwable th) {
    System.out.println("Failed from " +
        "asyncCmdClientHelper.processCommandParameters(cmd).");
    th.printStackTrace();
    System.exit(-1);
}

// Call the asynchronous command client to run the command.
asyncCmdClientHelper.execute(cmd);
System.out.println("\nCompleted running of command");

// Check the command result.
CommandResult result = cmd.getCommandResult();
if (result != null) {
    if (result.isSuccessful()) {
       System.out.println("\nCommand ran successfully "
                      + "with result\n" + result.getResult());
    }
    else {
        System.out.println("\nCommand ran with " +
```

```
                                        "Exception");
                    result.getException().printStackTrace();
                  }
              }
          } catch (Exception e) {
              e.printStackTrace();
          }
      }
  }

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {
    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                          notification);
    }
}
```

### What to do next

You can use the asset information that you viewed to perform other tasks. For instance, you might edit the asset to make improvements to the asset. You might export the asset and then import it into another configuration repository. You can then add the asset as a composition unit to a business-level application.

## Viewing a business-level application using programming

You can view business-level application information such as the description so that you can do other tasks associated with the business-level application, such as editing the business-level application. A business-level application is an administrative model that captures the entire definition of an enterprise-level application.

### Before you begin

This task assumes a basic familiarity with command framework programming. Read about command framework programming in the application programming interface documentation.

Before you can view a business-level application, you must have created the business-level application.

You can view a business-level application using programming, the administrative console, or the wsadmin tool.

### About this task

You must provide the blaID parameter to specify the business-level application that you are viewing.

Perform the following tasks to view a business-level application using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.

4. Create the asynchronous command client.

   An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Use the command manager that you created in a previous step to create and set up the command to view a business-level application.

   The command name is viewBLA. Use the required blaID parameter to specify the business-level application that you are viewing.

6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.

7. Display the command step.
8. Call the asynchronous command client to run the command to view a business-level application.

   You could have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.

9. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, you can view a business-level application.

## Example

The following example shows how to view a business-level application based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.cmdframework.CommandStep;
import com.ibm.websphere.management.cmdframework.TaskCommand;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class EditBLA {

    public static void main(String[] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local command manager.
            // Comment out the following lines to get the soapClient soap client if
            // you are going to use the local command manager. You would
```

```
// comment out the lines to and including
// CommandMgr cmdMgr =
// CommandMgr.getClientCommandMgr(soapClient);

String host = "localhost"; // Change to your host if it is not localhost.
String port = "8880"; // Change to your port number if it is not 8880.

Properties config = new Properties();
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
       AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
                  AdminClientFactory.createAdminClient(config);

// Create the command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager.
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();

System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required.
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
// This example creates a new session. You can replace the
// code below to use an existing session that has been
// created.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace the listener with
// null for the following AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
                AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command.
String cmdName = "viewBLA";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // View an existing
                               // business-level application
                               // using the session created.
System.out.println("\nCreated " + cmdName);


// Set the required blaID parameter.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
String blaID = "bla1";  // Replace the bla1 value with your value.
cmd.setParameter("blaID", blaID);
```

```
            System.out.println("\nSet blaID parameter to "
                    + cmd.getParameter("blaID"));

            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                       "parameters");
            } catch (Throwable th) {
                System.out.println("Throwing an exception from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Display step data.
            String[] stepNames = ((TaskCommand) cmd).listCommandSteps();
            for (int i = 0; i < stepNames.length; i++) {

                // Get the step.
                CommandStep step =
                           ((TaskCommand)cmd).gotoStep(stepNames[i]);

                List paramNames = step.listParameterName();

                System.out.println("----------- Step: " + step.getName() +
                                   " ----------");
                // Get the parameter values for each row.
                for (int j = 0; j < step.getNumberOfRows(); j++) {
                    System.out.println("  Row " + j);

                    for (int k = 0; k < paramNames.size(); k++)
                        System.out.println("      " + paramNames.get(k) +
                               ": " + step.getParameter(
                                      (String) paramNames.get(k), j));

                }

            }

            // Run the command to view the business-level application.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted command execution");

            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand executed successfully "
                           + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand executed with " +
                                           "Exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;
```

```
public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification.
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

### What to do next

You can use the information that you viewed about the business-level application to perform other tasks. You might edit the business-level application to make improvements to it. You might start and stop a business-level application, delete a business-level application, add a composition unit to a business-level application, and so on.

## Listing control operations using programming

You can list the control operations of a business-level application or a composition unit for a session. You can use control operations, such as start or stop, to change or query the runtime environment of a business-level application or a composition unit.

### Before you begin

Before you can list control operations of a business-level application or a composition unit for a session, you must have created an empty business-level application, imported an asset, and added a composition unit.

### About this task

You can list control operations of a business-level application or a composition unit using programming, the administrative console, or the wsadmin tool. This topic describes how to list control operations using programming.

To list control operations for a business-level application of a session, provide a blaID paramaeter value, but no cuID parameter value. To list control operations for a composition unit, specify both a blaID parameter value and a cuID parameter value. To list all control operations for the specified business-level application or the specified composition unit, do not specify an opName parameter value. To list the details for a specific control operation, set the opName parameter value to the name of the operation to list. To list details of the control operation definition, set the long parameter to true.

Perform the following tasks to list control operations for a business-level application or a composition unit of a session using programming.

1. Connect to the application server.

   The command framework allows the administrative command to be created and run with or without being connected to the application server. This step is optional if the application server is not running.
2. Create the command manager.

   The command manager provides the functionality to create a new administrative command or query existing administrative commands.
3. Optionally create the asynchronous command handler for listening to command notifications.

   Business-level application commands are implemented as asynchronous commands. To monitor the progress of the running command, you have to create an asynchronous command handler to receive notifications that the command generates.
4. Create the asynchronous command client.

An asynchronous command client provides a higher level interface to work with an asynchronous command. If you created an asynchronous command handler in the previous step, the handler is passed to the asynchronous command client. The asynchronous command client forwards the command notification to the handler and helps to control running of the command.

5. Create and set up the command that lists control operations of a business-level application or a composition unit of a session.
6. Call the processCommandParameters method in the asynchronous command client to process the command parameters.

   The command framework asynchronous command model requires this call.
7. Call the asynchronous command client to list the control operations of a business-level application or a composition unit of a session.

   You might have created an asynchronous command handler to implement the AsyncCommandHandlerIF interface class in a previous step. If you did, the asynchronous command client listens to command notifications and forwards the notifications to the handler. The handler performs any necessary actions while waiting for the command to complete.
8. Check the command result when the command completes.

   When the command finishes running, control is returned to the caller. You can then check the result by calling the command.getCommandResult method.

## Results

After you successfully run the code, a control operations of a business-level application or a composition unit for a session is displayed.

## Example

The following example shows how to list the control operation of a business-level application or a composition unit of a session based on the previous steps. Some statements are split on multiple lines for printing purposes.

```
package com.ibm.ws.management.application.task;

import java.util.Properties;

import com.ibm.websphere.management.AdminClient;
import com.ibm.websphere.management.AdminClientFactory;
import com.ibm.websphere.management.Session;
import com.ibm.websphere.management.cmdframework.AdminCommand;
import com.ibm.websphere.management.cmdframework.CommandMgr;
import com.ibm.websphere.management.cmdframework.CommandResult;
import com.ibm.websphere.management.async.client.AsyncCommandClient;

public class listControlOps {

    public static void main(String[] args) {

        try {

            // Connect to the application server.
            // This step is optional if you use the local
            // command manager. Comment out the lines to and including
            // CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(
            // soapClient);
            // to get the soapClient soap client if you use the local
            // command manager.
            String host = "localhost";
            String port = "8880"; // Change to your port number if
                                  // it is not 8880.

            Properties config = new Properties();
```

```
config.put(AdminClient.CONNECTOR_HOST, host);
config.put(AdminClient.CONNECTOR_PORT, port);
config.put(AdminClient.CONNECTOR_TYPE,
                  AdminClient.CONNECTOR_TYPE_SOAP);
System.out.println("Config: " + config);
AdminClient soapClient =
                  AdminClientFactory.createAdminClient(config);

// Create the command manager.
CommandMgr cmdMgr = CommandMgr.getClientCommandMgr(soapClient);

// Comment out the previous lines to create a client command
// manager if you are using a local command manager.
// Uncomment the following line to create a local command
// manager:
//
// CommandMgr cmdMgr = CommandMgr.getCommandMgr();
System.out.println("\nCreated command manager");

// Optionally create an asynchronous command handler.
// Comment out the following line if no further handling
// of command notification is required:
AsyncCmdTaskHandler listener = new AsyncCmdTaskHandler();

// Create an asynchronous command client.

// Set up the session.
String id = Long.toHexString(System.currentTimeMillis());
String user = "content" + id;
Session session = new Session(user, true);

// If no command handler is used, replace listener with
// null for the AsyncCommandClient object.
AsyncCommandClient asyncCmdClientHelper = new
                   AsyncCommandClient(session, listener);
System.out.println("\nCreated async command client");

// Create the command that lists the control operations.
String cmdName = "listControlOps";
AdminCommand cmd = cmdMgr.createCommand(cmdName);
cmd.setConfigSession(session); // List all the control operations
                              // using the session created.
System.out.println("\nCreated " + cmdName);

// Set the blaID parameter, which is required.
// The blaID is for either the business-level application whose control
// units you are listing or for the business-level application whose
// composition unit control operations you are listing.
// Change the blaID parameter according to your
// scenario.
// Examples of valid formats for the blaID parameter are:
// - bName
// - blaname=bName
// - WebSphere:blaname=bName
// This parameter accepts an incomplete ID as long as the incomplete
// ID can resolve to a unique business-level application.
// String blaID = "bla1";
// cmd.setParameter("blaID", blaID);

// System.out.println("\nSet blaID parameter to "
//                      + cmd.getParameter("blaID"));


// Optionally set the cuID parameter to the composition
// unit whose control operations you are listing.
// Examples of valid formats for the cuID parameter are:
// - name
```

```
            // - cuname=name
            // - WebSphere:cuname=name
            // This parameter accepts an incomplete ID as long as the
            // incomplete ID can resolve to a unique composition unit
            // within the business-level application.
            //
            // String cuID = "test5.zip";
            // cmd.setParameter("cuID", cuID);

            // System.out.println("\nSet cuID parameter to "
            //                       + cmd.getParameter("cuID"));

            // Optionally set the opName parameter of the operation to list.
            // String opName = "opName1";
            // cmd.setParameter("opName", opName);

            // System.out.println("\nSet opnameID parameter to "
            //                      + cmd.getParameter("opName"));

            // Optionally include details of the control operation definition
            // by setting the long parameter to true.
            // String long = "true";
            // cmd.setParameter("long", long);

            // System.out.println("\nSet long parameter to "
            //                       + cmd.getParameter("long"));


            // Call the asynchronous client helper to process parameters.
            try {
                asyncCmdClientHelper.processCommandParameters(cmd);
                System.out.println("\nCompleted process command " +
                                            "parameters");
            } catch (Throwable th) {
                System.out.println("Failed from " +
                    "asyncCmdClientHelper.processCommandParameters(cmd).");
                th.printStackTrace();
                System.exit(-1);
            }

            // Run the command to list control operations.
            asyncCmdClientHelper.execute(cmd);
            System.out.println("\nCompleted running of command");

            // Check the command result.
            CommandResult result = cmd.getCommandResult();
            if (result != null) {
                if (result.isSuccessful()) {
                    System.out.println("\nCommand ran successfully "
                                   + "with result\n" + result.getResult());
                }
                else {
                    System.out.println("\nCommand ran with " +
                                          "Exception");
                    result.getException().printStackTrace();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}


package com.ibm.ws.management.application.task;

import com.ibm.websphere.management.cmdframework.provider.CommandNotification;
```

```
import com.ibm.websphere.management.async.client.AsyncCommandHandlerIF;

public class AsyncCmdTaskHandler implements AsyncCommandHandlerIF {

    public void handleNotification(CommandNotification notification) {
        // Add your own code here to handle the received notification
        System.out.println("\nEXAMPLE: notification received: " +
                           notification);
    }
}
```

## What to do next

You can complete other tasks associated with business-level applications and composition units, such as deleting, starting, or stopping business-level applications or adding or exporting a composition unit.

# Chapter 9. Troubleshooting deployment

When you are having problems deploying an application, perform some basic diagnostics and verify your system's configuration to solve the problem.

- Select the problem you are having with deploying or installing developed code for WebSphere Application Server.
  - Errors or problems deploying, installing, or promoting applications
- To troubleshoot other deployment issues, use the following resources.
  - For current information available from IBM Support on known problems and their resolution, see the IBM Support page.
  - IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.
  - If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see Troubleshooting help from IBM.

## Application deployment problems

You might encounter problems when deploying, installing, or promoting applications. This topic suggests ways to resolve the problems.

What kind of problem are you having?

- "I installed my application using the wsadmin tool, but the application does not display under Applications > Application Types > WebSphere enterprise applications" on page 280
- "Unable to save a deployed application" on page 280
- "WASX7015E error running wsadmin command $AdminApp installInteractive or $AdminApp install" on page 280
- "Cannot install a CMP or BMP entity bean in an EJB 3.0 module" on page 280
- "Data definition language (DDL) generated by an assembly tool throws SQL error on target platform" on page 281
- "Error message ADMA0004E: Validation error in task Specifying the Default Datasource for EJB Modules returned when installing application using the administrative console or the wsadmin tool" on page 281
- "Cannot load resource WEB-INF/ibm-web-bnd.xmi in archive file" on page 282
- "″Timeout!!!″ error displays when attempting to install an enterprise application in the administrative console " on page 282
- "I get a NameNotFoundException message when deploying an application that contains an EJB module" on page 282
- "I get compilation errors and EJB deploy fails when installing an EJB JAR file generated for Version 5.x or earlier" on page 282
- "After installing the application onto a different machine, the application does not run" on page 283
- "A single file replaces all application files during application update" on page 283

Check the following first:

- Verify that the logical name that you have specified to appear on the console for your application, enterprise bean module or other resource does not contain invalid characters such as these: `- / \ : * ? ″ < > |`.
- If the application was installed using the wsadmin $AdminApp install command with the **-local** flag, restart the server or rerun the command without the `-local` flag.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, check to see if the problem is identified and documented.

© IBM Corporation 2004

## I installed my application using the wsadmin tool, but the application does not display under Applications > Application Types > WebSphere enterprise applications

The application might be installed but you have not saved the configuration:

1. Verify that the application subdirectory is located under the *app_server_root*/installedApps directory.
2. Run the $AdminApp list command and verify that the application is not among those displayed.
   - In the bin directory, run the wsadmin.bat or wsadmin.sh command.
   - From the wsadmin prompt, enter `$AdminApp list` and verify that the problem application is not among the items that display.
3. Reinstall your application using the wsadmin tool. Run the $AdminConfig save command in the wsadmin tool before exiting.

## Unable to save a deployed application

If you are unable to save a deployed application, the problem might be that too many files are opened, exceeding the limit of the operating system.

Only root has authority to adjust the maximum number of files for each process. Complete the following steps to modify the application to close files with disciplines:

1. After you open a file and complete your work, call the close method of the file to release the file handle back to the operating system.
2. Using the java.io.FileInputStream and the FileOutputStream classes as examples, you can invoke their close method to release any system resources that are associated with the stream.

## WASX7015E error running wsadmin command `$AdminApp installInteractive` or `$AdminApp install`

This problem has two possible causes:

- If the full text of the error is similar to:

  ```
  WASX7015E: Exception running command:
  "$AdminApp installInteractive C:/Documents and Settings/
    myUserName/Desktop/MyApp/myapp.ear";
  exception information:
  com.ibm.bsf.BSFException: error while
  eval'ing Jacl expression: can't find method "installInteractive"
  with 3 argument(s) for class
  "com.ibm.ws.scripting.AdminAppClient"
  ```

  The file and path name are incorrectly specified. In this case, since the path included spaces, it was interpreted as multiple parameters by the wsadmin program.

  Enter the path of the `.ear` file correctly. In this case, by enclosing it in double quotes:

  ```
  $AdminApp installInteractive "C:\Documents
  and Settings\myUserName\Desktop\MyApps\myapp.ear"
  ```
- If the full text of the error is similar to:

  ```
  WASX7015E: Exception running command: "$AdminApp installInteractive c:\MyApps\myapp.ear ";
  exception information: com.ibm.ws.scripting.ScriptingException:  WASX7115E:
  Cannot read input file
  "c:\WebSphere\AppServer\bin\MyAppsmyapp.ear"
  ```

  The application path is incorrectly specified. In this case, you must use ″forward-slash″ (/) separators in the path.

## Cannot install a CMP or BMP entity bean in an EJB 3.0 module

When installing an EJB 3.0 module that contains a container-managed persistence (CMP) or bean-managed persistence (BMP) entity bean, the installation fails.

The product does not support installation of applications that have a CMP or BMP entity bean packaged in an EJB 3.0 module. You must package CMP or BMP entity beans in an EJB 2.1 or earlier module.

To resolve this problem:

1. Package the CMP or BMP entity beans in EJB 2.1 or earlier modules.
2. Try installing your application with the EJB 2.1 or earlier modules.

## Data definition language (DDL) generated by an assembly tool throws SQL error on target platform

If you receive SQL errors in attempting to execute data definition language (DDL) statements generated by an assembly tool on a different platform, for example if you are deploying a container-managed persistence (CMP) enterprise bean designed on Windows onto a UNIX® operating system server, try the following actions:

- Browse the DDL statements for dependencies on specific user identifiers and passwords, and correct as necessary.
- Browse the DDL statements for dependencies on specific server names, and correct as necessary.
- Refer to the message reference of the vendor for causes and suggested actions regarding specific SQL errors. For IBM DB2, you can view the message references online at http://www.ibm.com/cgi-bin/db2www/data/db2 /udb/winos2unix/support/index.d2w/report.

If you receive the following error after executing a DDL file created on the Windows operating system or on operating systems such as AIX® or Linux, the problem might come from a difference in file formats:

```
SQL0104N  An unexpected token "CREATE TABLE AGENT  (COMM DOUBLE,  PERCENT  DOUBLE,  P"
was found following "        ".  Expected tokens may include:  " ".
SQLSTATE=42601
```

To resolve this problem:

- Use EDTF to edit the file.

## Error message `ADMA0004E: Validation error in task Specifying the Default Datasource for EJB Modules` returned when installing application using the administrative console or the wsadmin tool

If you see the following error when trying to install an application through the administrative console or the wsadmin command prompt:

```
AppDeploymentException: [ADMA0014E: Validation failed.
ADMA0004E: Validation error in task Specifying the Default Datasource for
EJB Modules  JNDI name is not
specified for module beannameBean Jar with URI filename.jar,META-INF/ejb-jar.xml.
You have not specified the
data source for each CMP bean belonging to this module. Either specify the data
source for each CMP beans or
specify the default data source for the entire module.]
```

one possible cause is that, in WebSphere Application Server Version 4.0, it was mandatory to have a data source defined for each CMP bean in each JAR. In Version 5.0 and later releases, you can specify either a data source for a container-managed persistence (CMP) bean or a default data source for all CMP beans in the JAR file. Thus during installation interaction, such as the installation wizard in the administrative console, the data source fields are optional, but the validation performed at the end of the installation checks to see that at least one data source is specified.

To correct this problem, step through the installation again, and specify either a default data source or a data source for each CMP-type enterprise bean.

If you are using the wsadmin tool, use the **`$AdminApp installInteractive`** ***`filename`*** command to receive prompts for data sources during installation, or to provide them in a response file.

Specify data sources as an option to the **$AdminApp install** command.

## Cannot load resource WEB-INF/ibm-web-bnd.xmi in archive file

The Web application tmp.war installs on WebSphere Application Server Versions 5.0 and 5.1, but fails on a WebSphere Application Server Version 6.0 or later server. The application fails to install because the WEB-INF/ibm-web-bnd.xmi file contains xmi tags that the underlying WCCM model no longer recognizes.

The following error messages display:

```
IWAE0007E Could not load resource "WEB-INF/ibm-web-bnd.xmi" in archive "tmp.war"
[2/24/05 14:53:10:297 CST] 000000bc SystemErr     R
AppDeploymentException:
com.ibm.etools.j2ee.commonarchivecore.exception.ResourceLoadException:
IWAE0007E Could not load resource "WEB-INF/ibm-web-bnd.xmi" in archive "tmp.war"
[2/24/05 14:53:10:297 CST] 000000bc SystemErr     R
com.ibm.etools.j2ee.commonarchivecore.exception.ResourceLoadException:
IWAE0007E Could not load resource "WEB-INF/ibm-web-bnd.xmi" in archive "tmp.war"
!Stack_trace_of_nested_exce!
com.ibm.etools.j2ee.exception.WrappedRuntimeException: Exception occurred loading
WEB-INF/ibm-web-bnd.xmi
!Stack_trace_of_nested_exce!
```

To work around this problem, remove the xmi:type=EJBLocalRef tag from the ibm-web-bnd.xmi file. Removing this tag does not affect the application because the tag was previously used for matching the cross document reference type. The application now works for the WebSphere Application Server Version 5.1 and later releases.

## ″Timeout!!!″ error displays when attempting to install an enterprise application in the administrative console

This error can occur if you attempt to install an enterprise application that has not been deployed.

To correct this problem:
- Open the *file_name*.ear file in an assembly tool and then click **Deploy**. This action creates a file with a name like Deployed_*file_name*.ear.
- In the administrative console, install the deployed .ear file.

## I get a NameNotFoundException message when deploying an application that contains an EJB module

If you specify that the EJB deployment tool be run during application installation and the installation fails with a NameNotFoundException message, ensure that the input JAR or EAR file does not contain source files. If there are source files in the input JAR or EAR file, the EJB deployment tools runs a rebuild before generating the deployment code.

To work around this problem, either remove the source files or include all dependent classes and resource files on the class path. Otherwise, the source files or the lack of access to dependent classes and resource files might cause problems during rebuilding of your application on the server.

## I get compilation errors and EJB deploy fails when installing an EJB JAR file generated for Version 5.x or earlier

When installing an old application that uses EJB modules that were built to run on WebSphere Application Server Version 5.x or earlier, compilation errors result and EJB deploy fails. The EJB JAR file contains Java source for the old generated code. The old Java source was generated for Version 5.x or before but, when deployed to a WebSphere Application Server Version 6.x product, it is compiled using the Version 6.x runtime JAR files.

To work around this problem, remove all `.java` files from the application `.ear` file. After the Java source files are removed, you can deploy the application onto a server successfully.

### After installing the application onto a different machine, the application does not run

If your application uses application level resources, its application level node information must be correct for the application to run as expected.

When you add application level resources to an application and deploy the application onto a machine, ensure that the application level node information is correct. Otherwise, when you install the application onto a different machine, it is installed to the wrong location and the application does not run as expected.

You can update the application level node information using an assembly tool. Update the nodeName from deploymentTargets of the `deployment.xml` file under ibmconfig. Also, ensure that binariesURL from deployedObject of the `deployment.xml` file has the correct path.

### A single file replaces all application files during application update

If you select the **Replace or add a single file** option of the application update wizard and the currently deployed application consists of several files, specify the full path name of the file to be replaced or added for **Specify the path beginning with the installed application archive file to the file to be replaced or added**.

A full path name usually has the structure *directory_path/file_name* and resembles the following:

```
PriceChangeSession.jar/priceChangeSession/priceChangeSessionBean.class
```

Do not specify less than the full path name for **Specify the path beginning with the installed application archive file to the file to be replaced or added**. For example, do not specify only a directory path:

```
PriceChangeSession.jar/priceChangeSession
```

If you specify less than a full path name, all files in the directory of the currently deployed application might be replaced by the single new file that was specified under **Specify the path to the file**.

## Application deployment troubleshooting tips

When you first test or run a deployed application, you might encounter problems.

Select the problem you are having with testing or the first run of deployed code for WebSphere Application Server:

- Server startup problems.
- "Application startup problems" on page 289.
- "Web resource is not displayed" on page 292.
- Data access problems.
- Enterprise bean cannot be accessed from a servlet, a JSP file, a stand-alone program, or another client.
- Application access problems.
- Access problems after enabling security.
- Security enablement followed by errors.
- Secure Sockets Layer errors.
- Application client sending SOAP request receives errors.
- "A client program does not work" on page 284.
- WebSphere MQ connection and queue connection factory creation errors.

You can use the following administrative console panels to inspect the configuration of your applications and JMS resources:

- For a view of the JMS resources for a given application, see the following panel: Messaging resources for this application.
- For a view of the applications and JMS resources for a given default messaging provider destination, see the following panel: Application resources for this destination.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see Troubleshooting help from IBM.

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

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

## A client program does not work

What kind of problem are you seeing?

### ActiveX client fails to display ASP files, or WebSphere Application Server resources (JSP files, servlet, or HTML pages) or both

A possible cause of this problem is that both IIS for serving Active Server Pages (ASP) files and an HTTP server that supports WebSphere Application Server (such as IBM HTTP Server) are deployed on the same host. This deployment leads to misdirected HTTP traffic if both servers are listening on the same port (such as the default port 80).

To resolve this problem, either:
- Open the IIS administrative panel, and edit the properties of the default Web server to change the port number to a value other than 80
- Install IIS and the HTTP server on separate servers.

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

### Plants by WebSphere Catalog Manager (pbwsCatalogMgr) exceptions

When you federate a standalone server into a Deployment Manager cell, the bootstrap port number of the application server may change. This will cause the client to not be able to communicate with the server, thus causing an exception. The following scenario may cause an exception when you start Plants by WebSphere:
1. Install a standalone WebSphere Application Server.
2. Run the Plants by WebSphere example.
3. Create a Deployment Manager (DMGR) using the Profile Management tool or by using the **manageprofiles** command.
4. Federate the standalone WebSphere Application Server into a Deployment Manager cell using the **addNode** command.
5. Start **pbwsCatalogMgr**.

To avoid the exception, locate the new (changed) port number on the server and modify the client configuration to match the port number on the server.
1. Go to `*was_server_root*\profiles*your_server_name*\config\cells*your_cell*\nodes*your_node*`.
   a. Open the `serverindex.xml` file.
   b. Locate the BOOTSTRAP_ADDRESS port number of the application server, for example `9810`.

2. Assign this port number to the client to communicate with your newly-federated application server. Go to `was_client_root\bin` and edit the `setupClient.bat` file.
3. Locate the line 'SET SERVERPORTNUMBER' and set the value for it to `9810`.

   If you have security enabled, ensure that the bus security is also enabled and that a user is defined to the bus connector role before running pbwsCatalogMgr.
4. Restart the node agent and the application server.

The client is now properly set up to start **pbwsCatalogMgr**.

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

## Application startup errors

Use this information for troubleshooting problems that occur when starting an application.

What kind of error do you see when you start an application?


- "HTTP server and Application Server are working separately, but requests are not passing from HTTP server to Application Server"
- "File serving problems" on page 286
- "Graphics do not appear in the JSP file or servlet output" on page 286
- "SRVE0026E: [Servlet Error]-[Unable to compile class for JSP file" on page 287
- "After modifying and saving a JSP file, the change does not show up in the browser (the old JSP file displays)" on page 288
- "Message like ″Message: /jspname.jsp(9,0) Include: Mandatory attribute page missing″ appears when attempting to browse JSP file" on page 288
- "The Java source generated from a JSP file is not retained in the temp directory (only the class file is found)" on page 288
- "The JSP Batch Compiler fails with the message ″Enterprise Application [application name you typed in] not found.″" on page 289
- "There is a translation problem with non-English browser input" on page 289
- "Scroll bars do not appear around items in the browser window" on page 289
- "Error ″Page cannot be displayed... server not found or DNS error″ appears when attempting to browse a JavaServer Pages (JSP) file using Internet Explorer" on page 289


### HTTP server and Application Server are working separately, but requests are not passing from HTTP server to Application Server

If your HTTP server appears to be functioning correctly, and the Application Server also works on its own, but browser requests sent to the HTTP server for pages are not being served, a problem exists in the WebSphere Application Server plug-in.

In this case:

1. Determine whether the HTTP server is attempting to serve the requested resource itself, rather than forwarding it to the WebSphere Application Server.
   a. Browse the HTTP server access log (*IHS install root*/`logs/access.log` for IBM HTTP Server). It might indicate that it could not find the file in its own document root directory.
   b. Browse the plug-in log file as described below.
2. Refresh the plugin-cfg.xml file that determines which requests sent to the HTTP server are forwarded to the WebSphere Application Server, and to which Application Server.

   Use the console to refresh this file:
   - In the WebSphere Application Server administrative console, expand the Environment tree control.
   - Click **Update WebSphere Plugin**.
   - Stop and restart the HTTP server.

If you are using IBM HTTP Server for iSeries or Lotus® Domino® for iSeries, you do not need to restart the HTTP server.
- Retry the Web request.

If you have created a Web server definition to model your Web server instance, the file is located under *profile_root*/config/cells/*cell_name*/nodes/*Web_server_node_name*/servers/*Web_server_name*. If you have not, the file is located under *profile_root*/config/cells.

3. Browse the *plugin_install_root*/logs/*web_server_name*/http_plugin.log file for clues to the problem. Make sure the timestamps with the most recent plug-in information stanza, which is printed out when the plug-in is loaded, correspond to the time the Web server started.
4. Turn on plug-in tracing by setting the LogLevel attribute in the plugin-cfg.xml file to Trace and reloading the request. Browse the *plugin_install_root*/logs/*Web_server_name*/http_plugin.log file. You should be able to see the plug-in attempting to match the request URI with the various URI definitions for the routes in the plugin-cfg.xml. Check which rules the plug-in is not matching against and then figure out if you need to add additional ones. If you just recently installed the application you might need to manually regenerate the plug-in configuration to pick up the new URIs related to the new application.

For further details on troubleshooting plug-in-related problems, see Webserver plug-in troubleshooting tips located in the *Administering applications and their environment* PDF book.

## File serving problems

If text output appears on your JSP- or servlet-supported Web page, but image files do not:

- Verify that your files are in the right place: the **document root** directory of your Web application WebSphere Application Server follows the J2EE standard, which means that the document root is the *Web_module_name*.war directory of your deployed Web application.

  Typically this directory will be found in the *profile_root*/installedApps/*nodename*/*appname*.ear directory or *profile_root*/installedApps/*nodename*/*appname*Network.ear directory.

  If the files are in a subdirectory of the document root, verify that the reference to the file reflects that. That is, if the invoices.html file is stored in Windows directory *Web_module_name*.war\invoices, then links from other pages in the Web application to display it should read "invoices\invoices.html", not "invoices.html".
- Verify that your Web application is configured to enable file serving (in other words, that it is enabled to display static resources like image and .html files):
  1. View the file serving property of the hosting Web module by browsing the source .war file in an assembly tool. If necessary, update the property and redeploy the module. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.
  2. Edit the **fileServingEnabled** property in the deployed Web application ibm-web-ext.xmi configuration file.

     The file typically is found in the *profile_root*/config/cells/*nodename* or *nodename*Network/applications/*application_name*/deployments/*application name*/*Web_module_name*/web-inf directory.

## Graphics do not appear in the JSP file or servlet output

If text output appears on your JSP- or -servlet-supported Web page, but image files do not:

- Verify that your graphic files are in the right place: the **document root** directory of your Web application. WebSphere Application Server Version 5 follows the J2EE standard, which means that the document root is the *Web_module_name*.war directory of your deployed Web application.

  Typically, this directory is found in the *profile_root*/installedApps/*nodename*/*appname*.ear directory or *profile_root*/installedApps/*nodename*/*appname*Network.ear directory.

If the graphics files are in a subdirectory of the document root, verify that the reference to the graphic reflects that; for example, if the `banner.gif` file is stored in Windows directory *Web_module_name*.war/ images, the tag to display it should read: **`<img SRC="images/banner.gif">`**, not `<img SRC="banner.gif">`.

- Verify that your Web application is configured to enable file serving (that is, display of static resources like image and `.html` files).
  1. View the file serving property of the hosting Web module by browsing the source `.war` file in an assembly tool. If necessary, update the property and redeploy the module. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.
  2. Edit the **`fileServingEnabled`** property in the deployed Web application `ibm-web-ext.xmi` configuration file.

     The file typically is found in the *`profile_root`*`/config/cells/`*`nodename`* or *`nodename`*`Network/ applications/`*`application_name`*`/deployments/`*`application name`*`/`*`Web_module_name`*`/web-inf` directory.
  3. After completing the previous step:
     - In the administrative console, expand the **Environment** tree control .
     - Click **Update WebSphere Plugin**.
     - Stop and restart the HTTP server and retry the Web request.

## SRVE0026E: [Servlet Error]-[Unable to compile class for JSP file

If this error appears in a browser when trying to access a new or modified .jsp file for the first time, the most likely cause is that the JSP file Java source failed (was incorrect) during the javac compilation phase.

Check the SystemErr.log file for a compiler error message, such as:

```
C:\WASROOT\temp\ ... test.war\_myJsp.java:14: \Duplicate variable declaration: int myInt was int myInt
int myInt = 122;
String myString = "number is 122";
static int myStaticInt=22;
int myInt=121;
      ^
```

Fix the problem in the JSP source file, save the source and request the JSP file again.

If this error occurs when trying to serve a JSP file that was copied from another system where it ran successfully, then there is something different about the new server environment that prevents the JSP file from running. Browse the text of the error for a statement like:

```
Undefined variable or class name: MyClass
```

This error indicates that a supporting class or jar file is not copied to the target server, or is not on the class path. Find the MyClass.class file, and place it on the Web module WEB-INF/classes directory, or place its containing .jar file in the Web module WEB-INF/lib directory.

Verify that the URL used to access the resource is correct by doing the following:

- For a JSP file, `html` file, or image file: **`http://`*`host_name/Web_module_context_root/subdir under doc root, if any/filename.ext`***. The document root for a Web application is the *`application_name`*`.WAR` directory of the installed application.
  - For example, to access the myJsp.jsp file, located in c:\WebSphere\ApplicationServer\installedApps\ myEntApp.ear\myWebApp.war\invoices on myhost.mydomain.com, and assuming the context root for the myWebApp Web module is myApp, the URL is `http://myhost.mydomain.com/myApp/invoices/ myJsp.jsp`.
  - JSP serving is enabled by default. File serving for HTML and image files must be enabled as a property of the Web module, in an assembly tool, or by setting the **`fileServingEnabled`** property to **true** in the `ibm-web-ext.xmi` file of the installed Web application and restarting the application. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.

- For servlets served by class name, the URL is `http://`*`hostname`*`/`*`Web_module_context_root`*`/servlet/`*`packageName.className`*.

  For example, to access myCom.myServlet.class, located in profile_root/installedApps/myEntApp.ear/myWebApp.war/WEB-INF/classes, and assuming the context root for the myWebApp module is "myApp", the URL would be `http://myhost.mydomain.com/myApp/servlet/myCom.MyServlet`.
- Serving servlets by class name must be enabled as a property of the Web module, and is enabled by default. File serving for HTML and image files must be enabled as a property of the Web application, in an assembly tool, or by setting the **`fileServingEnabled`** property to **true** in the **`ibm-web-ext.xmi`** file of the installed Web application and restarting the application. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.

Correct the URL in the "from" HTML file, servlet or JSP file. An HREF with no leading slash (/) inherits the calling resource context. For example:
- an HREF in `http://[hostname]/myapp/servlet/MyServlet` to `"ServletB"` resolves to `"http://hostname/myapp/servlet/ServletB"`
- an HREF in `http://[hostname]/myapp/servlet/MyServlet` to `"servlet/ServletB"` resolves to `"http://hostname/myapp/servlet/servlet/ServletB"` (an error)
- an HREF in `http://[hostname]/myapp/servlet/MyServlet` to `"/ServletB"` resolves to `"http://hostname/ServletB"` (an error, if `ServletB` requires the same context root as `MyServlet`)

### After modifying and saving a JSP file, the change does not show up in the browser (the old JSP file displays)

It is probable that the Web application is not configured for servlet reloading, or the reload interval is too high.

To correct this problem, in an assembly tool, check the **Reloading Enabled** flag and the **Reload Interval** value in the IBM Extensions for the Web module in question. Enable reloading, or if it is already enabled, then set the Reload Interval lower. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.

### Message like "Message: /jspname.jsp(9,0) Include: Mandatory attribute page missing" appears when attempting to browse JSP file

It is probable that the JSP file failed during the translation to Java phase. Specifically, a JSP directive, in this case an Include statement, was incorrect or referred to a file that could not be found.

To correct this problem, fix the problem in the JSP source, save the source and request the JSP file again.

### The Java source generated from a JSP file is not retained in the temp directory (only the class file is found)

It is probable that the JSP processor is not configured to keep generated Java source.

In an assembly tool, check the **JSP Attributes** under **Assembly Property Extensions** for the Web module in question. Make sure the **keepgenerated** attribute is there and is set to `true`. If not, set this attribute and restart the Web application. To see the results of this operation, delete the class file from the temp directory to force the JSP processor to translate the JSP source into Java source again. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.

### The JSP Batch Compiler fails with the message ″Enterprise Application [application name you typed in] not found.″

It is probable that the full enterprise application path and name, starting with the `.ear` subdirectory that resides in the `applications` directory is expected as an argument to the JspBatchCompiler tool, not just the display name.

The directory path is *profile_root*`/config/cells/`*node_name*`Network/applications`.

For example:
- ″`JspBatchCompiler -enterpriseapp.name sampleApp.ear/deployments/sampleApp`″ is correct, as opposed to
- ″`JspBatchCompiler -enterpriseapp.name sampleApp`″, which is incorrect.

### There is a translation problem with non-English browser input

If non-English-character-set browser input cannot be translated after being read by a servlet or JSP file, ensure that the request parameters are encoded according to the expected character set before reading. For example, if the site is Chinese, the target `.jsp` file should have a line:

```
req.setCharacterEncoding("gb2312");
```

before any req.getParameter method calls.

This problem affects servlets and `jsp` files ported from earlier versions of WebSphere Application Server, which converted characters automatically based upon the locale of the WebSphere Application Server.

### Scroll bars do not appear around items in the browser window

In some browsers, tree or list type items that extend beyond their allotted windows do not have scroll bars to permit viewing of the entire list.

To correct this problem, right-click on the browser window and click **Reload** from the menu.

### Error ″Page cannot be displayed... server not found or DNS error″ appears when attempting to browse a JavaServer Pages (JSP) file using Internet Explorer

This error can occur when an HTTP timeout causes the servant to be brought down and restarted. To correct this problem, increase the ConnectionIOTimeOut value:
1. From the administrative console, select **System administration > Deployment manager > Administration Services > Custom Properties**
2. Select ConnectionIOTimeOut.
3. Increase the ConnectionIOTimeOut value.
4. Click **OK**.

## Application startup problems

When an application is not starting or starting with errors, the problem could be from one of various sources.

What kind of error do you see when you start an application?
- A “java.lang.ClassNotFoundException: classname Bean_AdderServiceHome_04f0e027Bean” on page 290 error occurs
- A “ConnectionFac E J2CA0102E: Invalid EJB component: Cannot use an EJB module with version 1.1 using The Relational Resource Adapter” on page 290 error occurs

- “NMSV0605E: ″A Reference object looked up from the context...″ error when starting an application” on page 291.
- “A Page Not Found, Array Index Out of Bounds, or other error when an updated application restarts” on page 291

If none of these errors match the error you see:
- Browse the log files of the application server for this application looking for clues. By default, these files are: *app_server_root*/logs/*server_name*/SystemErr.log and SystemOut.log.
- Look up any error or warning messages in the message reference table by clicking the **Reference** view and expanding **Messages**.

If you do not see a problem that resembles yours, or if the information provided does not solve your problem, see Troubleshooting help from IBM.

## java.lang.ClassNotFoundException: *classname* Bean_AdderServiceHome_04f0e027Bean

An similar exception occurs when you try to start an undeployed application containing enterprise beans, or containing undeployed enterprise bean modules.

Enterprise JavaBeans modules created in an assembly tool intentionally have incomplete configuration information. Deploying these modules completes the configuration by reading the module’s deployment descriptor and completing platform- or installation-dependent settings and adding related classes to the Enterprise JavaBeans JAR file.

To avoid this problem, do the following:
- Use an assembly tool and administrative console to generate deployment code and install the application or Enterprise JavaBeans module onto a server.
  1. Uninstall the application or Enterprise JavaBeans module in the administrative console.
  2. Configure your assembly tool so the target server is a WebSphere Application Server installation such as **WebSphere Application Server v6**. If you do not have access to the target server, you can specify a false location such as c:\temp. Specifying a false location enables you to assemble and generate deployment code for the enterprise bean.
  3. In the Project Explorer view of an assembly tool, right-click the enterprise bean (Enterprise JavaBeans) in the undeployed .ear file containing the Enterprise JavaBeans module or the standalone undeployed Enterprise JavaBeans JAR file, and click **Deploy**. If your assembly tool can access the WebSphere Application Server target server, deployment code is generated for the Enterprise JavaBeans and the assembly tool attempts to install the application or module onto the target server. If your assembly tool cannot access the WebSphere Application Server target server or the installation fails, use the deployment code that is generated for the next step.

     For information on using an assembly tool, refer to Chapter 3, “Assembling applications,” on page 11.
  4. Use the wsadmin $AdminApp install command or the administrative console to install the deployed version created by the assembly tool.
- If you use the wsadmin $AdminApp install command, uninstall it and then reinstall using the -EJBDeploy option. Follow the install command with the $AdminConfig save command.

## ConnectionFac E J2CA0102E: Invalid EJB component: Cannot use an EJB module with version 1.1 using The Relational Resource Adapter

This error occurs when an enterprise bean developed to the Enterprise JavaBeans 1.1 specification is deployed with a WebSphere Application Server V5 J2C-compliant data source, which is the default data source. By default, persistent enterprise beans created under WebSphere Application Server V4.0 using

the Application Assembly Tool fulfill the Enterprise JavaBeans 1.1 specification. To run on WebSphere Application Server V6, these enterprise beans must be associated with a WebSphere Application Server V4.0-type data source.

Either modify the mapping in the application of enterprise beans to associate 1.x container managed persistence (CMP) beans to associate them with a V4.0 data source or delete the existing data source and create a V4.0 data source with the same name.

To modify the mapping in the application of enterprise beans, in the WebSphere Application Server administrative console, select the properties for the problem application and use **map resource references to resources** or **Map data sources for all 1.x CMP beans** to switch the data source the enterprise bean uses. Save the configuration and restart the application.

To delete the existing data source and create a V4.0 data source with the same name:
1. In the administrative console, click **Resources > Manage JDBC Providers >** *JDBC_provider_name* **> Data sources**.
2. Delete the data source associated with the Enterprise JavaBeans 1.1 module.
3. Click **Resources > Manage JDBC Providers >** *JDBC_provider_name* **> Data sources (Version 4)**.
4. Create the data source for the Enterprise JavaBeans 1.1 module.
5. Save the configuration and restart the application.

## NMSV0605E: ″A Reference object looked up from the context...″ error when starting an application

If the full text of the error is similar to:

```
[7/17/02 15:20:52:093 CDT] 5ae5a5e2 UrlContextHel W NMSV0605E:
   A Reference object looked up from the context
   "java:" with the name "comp/PM/WebSphereCMPConnectionFactory" was sent to the JNDI Naming Manager
   and an exception resulted. Reference data follows:
   Reference Factory Class Name: com.ibm.ws.naming.util.IndirectJndiLookupObjectFactory
   Reference Factory Class Location URLs:
   Reference Class Name: java.lang.Object
   Type: JndiLookupInfo
   Content: JndiLookupInfo: ; jndiName="eis/jdbc/MyDatasource_CMP"; providerURL="";
       initialContextFactory=""
```

then the problem might be that the data source intended to support a CMP enterprise bean is not correctly associated with the enterprise bean.

To resolve this problem:
1. Select the **Use this Data Source in container managed persistence (CMP)** check box in the data source ″General Properties″ panel of the administrative console.
2. Verify the JNDI name in either of the following ways:
   - Verify that the JNDI name given in the administrative console under **Resources > Manage JDBC Provider > DataSource > JNDI Name** for DataSource matches the JNDI name given for CMP or BMP resource bindings at the time of assembling the application in an assembly tool.
   - Check the JNDI name for CMP or BMP resource bindings specified in the code by J2EE application developer. Open the deployed .ear folder in an assembly tool, and look for the JNDI name for your entity beans under CMP or BMP resource bindings. Verify that the names match.

## A Page Not Found, Array Index Out of Bounds, or other error when an updated application restarts

If an application is updated while it is running, WebSphere Application Server automatically stops the application or only its changed components, updates the application logic, and restarts the stopped application or its components. For more information on the restarting of updated applications, refer to

Fine-grained recycle behavior in *IBM WebSphere Developer Technical Journal: System management for WebSphere Application Server V6 -- Part 5 Flexible options for updating deployed applications*.

A Page Not Found, Array Index Out of Bounds, or other error might occur during restarting.

To minimize the occurrence of such errors, update applications in a test environment before updating the applications in a production environment. Do not put changes directly into a production environment.

## Web resource is not displayed

Use this information to troubleshoot problems that occur when attempting to display a resource in a browser.

If you are not able to display a resource in your browser, follow these steps:

1. Verify that your HTTP server is healthy by accessing the URL `http://server_name` from a browser and seeing whether the Welcome page appears. This action indicates whether the HTTP server is up and running, regardless of the state of WebSphere Application Server.
2. If the HTTP server Welcome page does not appear, that is, if you get a browser message like `page cannot be displayed` or something similar, try to diagnose your Web server problem.
3. If the HTTP server appears to function correctly, the Application Server might not be serving the target resource. Try to access the resource directly through the Application Server instead of through the HTTP server.

   If you cannot access the resource directly through the Application Server, verify that the URL used to access the resource is correct.

   If the URL is incorrect and it is created as a link from another JavaServer Pages (JSP) file, servlet, or HTML file, try correcting it in the browser URL field and reloading, to confirm that the problem is a malformed URL. Correct the URL in the "from" HTML file, servlet or JSP file.

   If the URL appears to be correct, but you cannot access the resource directly through the Application Server, verify the health of the hosting Application Server and Web module:
   a. View the hosting Application Server and Web module in the administrative console to verify that they are up and running.
   b. Copy a simple HTML or JSP file, such as `SimpleJsp.jsp`, which is in the WebSphere Application Server directory structure, to your Web module document root, and try to access the file. If successful, the problem is with the resource.

      View the JVM log of your Application Server to find out why your resource cannot be found or served .
4. If you can access the resource directly through the Application Server, but not through an HTTP server, the problem lies with the HTTP plug-in, the component that communicates between the HTTP server and the WebSphere Application Server.
5. If the JSP file and the servlet output are served, but not static resources such as `.html` and image files, see the steps for enabling file serving.
6. If certain resources display correctly, but you cannot display a servlet by its class name:
   - Verify that the servlet is in a directory in the Web module class path, such as in the /*Web_module_name*.war/WEB-INF/classes directory.
   - Verify that you specify the full class name of the servlet, including its package name, in the URL.
   - Verify that "`/servlet`" precedes the class name in the URL. For example, if the root context of a Web module is "`myapp`", and the servlet is `com.mycom.welcomeServlet`, then the URL reads:

     `http://hostname/myapp/servlet/com.mycom.welcomeServlet`
   - Verify that serving the servlets by class name is enabled for the hosting Web module by opening the source Web module in an assembly tool and browsing the *serve servlets by classname* setting in the IBM Extensions property page. If necessary, enable this flag and redeploy the Web module. For more information about the assembly tool, refer to the assembly tools section of the *Developing and deploying applications* PDF book.
   - For servlets or other resources served by mapped URLs, the URL is http://*hostname*/*Web module context root*/*mappedURL*.

If none of these steps fixes your problem, see if the problem has been identified and documented by looking at available online support (hints and tips, technotes, and fixes). If you do not find your problem listed there, see Troubleshooting help from IBM.

## Diagnosing Web server problems

If you are unable to view the welcome page of your HTTP server, determine if the server is operating properly.

If the HTTP server does not start:

- Examine the HTTP server error log for clues.
- Try restoring the HTTP server to its configuration prior to installing WebSphere Application Server and restarting it. If you are using IBM HTTP Server:
  - Rename the file *IHS_install_dir*\httpd.conf.
  - Copy the httpd.conf.default file to the httpd.conf directory.
  - If Apache is running, stop and restart it.
- For the Sun ONE (iPlanet) Web server, restore the obj.conf configuration file for Sun ONE V4.1 and both obj.conf and magnus.conf files for Sun ONE V6.0 and later.
- For the Microsoft® Internet Information Server (IIS), remove the WebSphere Application Server plug-in through the IIS administrative GUI.

If restoring the HTTP server default configuration file works, manually review the configuration file that has WebSphere Application Server updates to verify directory and file names for WebSphere Application Server files. If you cannot manually correct the configuration, you can uninstall and reinstall WebSphere Application Server to create a clean HTTP configuration file.

If restoring the default configuration file does not help, contact technical support for the Web server you are using. If you are using IBM HTTP Server with WebSphere Application Server, check available online support (hints and tips, technotes, and fixes). If you do not find your problem listed there, see Troubleshooting help from IBM

## Accessing a Web resource through the application server and bypassing the HTTP server

You can bypass the HTTP server and access a Web resource through the application server. It is not recommended to serve a production Web site in this way, but it provides a good diagnostic tool when it is not clear whether a problem resides in the HTTP server, WebSphere Application Server, or the HTTP plug-in.

To access a Web resource through the Application Server:

1. Determine the port of the HTTP service in the target application server.
   a. In the WebSphere administrative console, click **Servers > Application Servers**.
   b. Select the target server, then under Additional Properties click **Web Container**.
   c. Under the Additional Properties of the Web container, click **HTTP Transports**. You see the ports listed for virtual hosts served by the application server.
   d. There can be more than one port listed. In the default application server (server1), for example, 9060 is the port reserved for administrative requests, 9443 and 9043 are used for SSL-encrypted requests. To test the sample "`snoop`" servlet, for example, use the default application port 9080, unless it changes.
2. Use the HTTP transport port number of the application server to access the resource from a browser. For example, if the port is 9080, the URL is `http://`*`hostname`*`:9080/myAppContext/myJSP.jsp`.
3. If you are still unable to access the resource, verify that the HTTP transport port is in the "Host Alias" list:
   a. Click **Application Servers > Your_ApplicationServer > Web Container > HTTP Transports** to check the Default virtual host and the HTTP transport ports used by this application server.

b. Click **Environment > Virtual Hosts > default host > Host Aliases** to check if the HTTP transport port exists. Add an entry if necessary. For example, if the HTTP port for your application is server is 9080, add a host alias of *:9082.

## Application uninstallation problems

When you try to uninstall an application or node, you might encounter problems. This topic suggests ways to resolve uninstallation problems.

What kind of problem are you having?

- After uninstalling an application through wsadmin tool, the application continues to run and throws ″DocumentIOException″

If none of these steps fixes your problem:

- Make sure that the application and its Web and EJB modules are in a stopped state before uninstalling.
- If you are uninstalling or installing an application using **wsadmin**, make sure that you are using the **`-conntype NONE`** option to invoke **wsadmin** and enable local mode. To use the **`-conntype NONE`** option, stop the hosting application server before uninstalling the application.
- Check to see if the problem has been identified and documented by looking at the available online support (hints and tips, technotes, and fixes).
- If you don't find your problem listed there contact IBM support

### After uninstalling application through the wsadmin tool, the application throws ″DocumentIOException″

If this exception occurs after the application was uninstalled using wsadmin with the `-conntype NONE` option:

- Restart the server or,
- Rerun the uninstall command without the `-conntype NONE` option.

# Chapter 10. Adding logging and tracing to your application

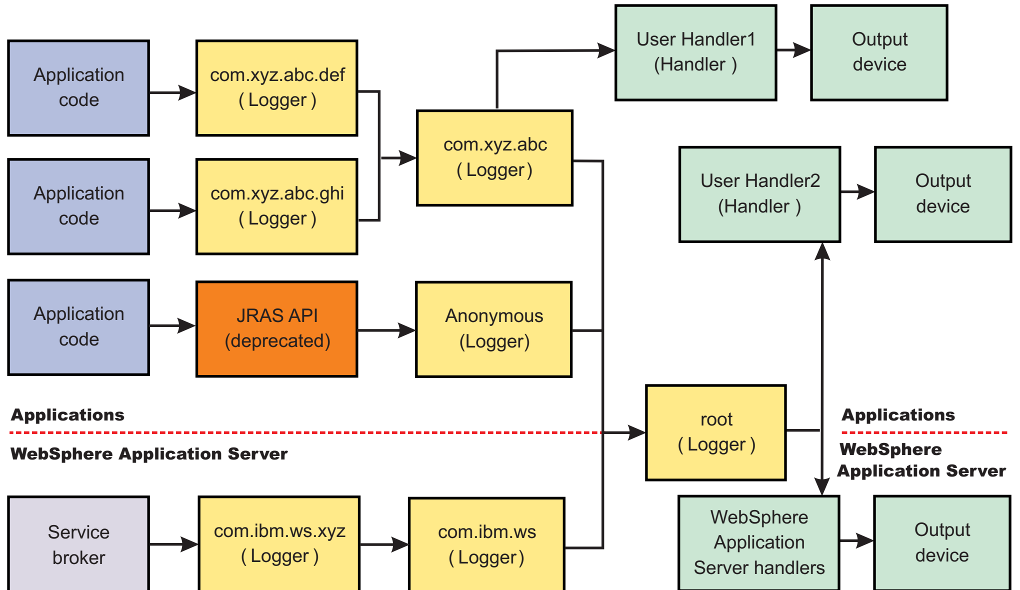
You can add logging and tracing to applications to help analyze performance and diagnose problems in WebSphere Application Server.

## About this task

**Deprecation:** The JRas framework that is described in this information center is deprecated. However, you can achieve the same results using Java logging.

Designers and developers of applications that run with or under WebSphere Application Server, such as servlets, JavaServer Pages (JSP) files, enterprise beans, client applications, and their supporting classes, might find it useful to use Java logging for generating their application logging.

This approach has advantages over adding `System.out.println` statements to your code:
- Your messages are displayed in the WebSphere Application Server standard log files, using a standard message format with additional data, such as a date and time stamp that are added automatically.
- You can more easily correlate problems and events in your own application to problems and events that are associated with WebSphere Application Server components.
- You can take advantage of the WebSphere Application Server log file management features.




1. Enable and configure one of the supported types of logging. Use one of the following methods:
   - “Configuring Java logging using the administrative console” on page 296
   - “Configuring applications to use Jakarta Commons Logging” on page 317
   - “Logging Common Base Events in WebSphere Application Server” on page 343.
2. Customize the properties to meet your logging needs. For example, enable or disable a particular log, specify the number of logs to be kept, and specify a format for log output.
3. Restart the application server after making static configuration changes.

## Configuring Java logging using the administrative console

Java logging provides a standard logging API for your applications. Before applications can log diagnostic information, you need to specify how you want the server to handle log output and what level of logging you require.

### About this task

Developing, deploying and maintaining applications are complex tasks. When an application encounters an unexpected condition, it might not be able to complete a requested operation. You might want the application to inform the administrator that the operation failed and tell the administrator why the operation failed. This information enables the administrator to take the proper corrective action. Application developers might need to gather detailed information that relates to the path of a running application to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as *logging* and *tracing*. For more information read "Java logging."

Using the administrative console, you can:
- Enable or disable a particular log, specify where log files are stored and how many log files are kept.
- Specify the level of detail in a log, and specify a format for log output.
- Set a log level for each logger.

You can change the log configuration statically or dynamically. Static configuration changes affect applications when you start or restart the application server. Dynamic or run time configuration changes apply immediately.

When a log is created, the level value for that log is set from the configuration data. If no configuration data is available for a particular log name, the level for that log is obtained from the parent of the log. If no configuration data exists for the parent log, the parent of that log is checked, and so on up the tree, until a log with a non-null level value is found. When you change the level of a log, the change is propagated to the children of the log, which recursively propagates the change to their children, as necessary.

1. Optional: See the Java documentation for the java.util.logging class for a full description of the syntax and the construction of logging methods.
2. Set the logging levels for your logs:
   a. In the navigation pane, click **Servers > Application Servers**.
   b. Click the name of the server that you want to work with.
   c. Under Troubleshooting, click **Logs and Trace**.
   d. Click **Change Log Detail levels**.
   e. To make a static change to the configuration, click the **Configuration** tab. A list of well-known components, packages, and groups is displayed. To change the configuration dynamically, click the **Runtime** tab. The list of components, packages, and groups displays all the components that are currently registered on the running server.
   f. Select a component, package, or group to set a logging level.
   g. Click **Apply**.
   h. Click **OK**.
3. To have static configuration changes take effect, stop then restart the application server.

## Java logging

Java logging is the logging toolkit that is provided by the java.util.logging package. Java logging provides a standard logging API for your applications.

Message logging (messages) and diagnostic trace (trace) are conceptually similar, but do have important differences. These differences are important for application developers to understand to use these tools properly. The following operational definitions of messages and trace are provided.

**Message**
: A message entry is an informational record that is intended for end users, systems administrators, and support personnel to view. The text of the message must be clear, concise, and interpretable by an end user. Messages are typically localized and displayed in the national language of the end user. Although the destination and lifetime of messages might be configurable, enable some level of message logging in normal system operation. Use message logging judiciously because of performance considerations and the size of the message repository.

**Trace**
: A trace entry is an information record that is intended for service engineers or developers to use. As such, a trace record might be considerably more complex, verbose, and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries can be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but can be enabled as needed to gather diagnostic information.

The application server redirects the system streams at the server startup. There is no way to allow the application to output logging to the console because the system streams can not be obtained by the application. If you would like to use console to monitor the application without using the console handler, you can either monitor the `SystemOut.log` file, or monitor a file created by another file handler.

**Note:** The application server uses Java logging internally and therefore certain restrictions apply for using system streams with this logging API by applications. During server startup, the standard output and error streams are replaced with special streams that write to the logging infrastructure, in order to include the output of the system streams in the log files. Because of this, applications can not use `java.util.logging.ConsoleHandler`, or any handler writing to `System.err` or `System.out` streams, attached to the root logger. If the user does attach the handler to the root logger, an infinite loop is created within the logging infrastructure, leading to stack overflow and server crash.

If the use of a handler that writes to system streams is necessary, attach it to a non-root logger so that it does not publish log records to parent handlers. The data written to the system streams is then formatted and written to the corresponding system stream log file. To monitor what is being written system streams, the configured log files (`SystemOut.log` and `SystemErr.log` by default) can be monitored.

## Log level settings

Use this topic to configure and manage log level settings.

Using log levels you can control which events are processed by Java logging. When you change the level for a logger, the change is propagated to the children of the logger.

**Change Log Detail Levels**

Enter a log detail level that specifies the components, packages, or groups to trace. The log detail level string must conform to the specific grammar described in this topic. You can enter the log detail level string directly, or generate it using the graphical trace interface.

If you select the Configuration tab, a static list of well-known components, packages, and groups is displayed. This list might not be exhaustive.

If you select the Runtime tab, the list of components, packages, and group are displayed with all the components that are registered on the running application server and in the static list.

The format of the log detail level specification is:

```
<component> = <level>
```

where <component> is the component for which to set a log detail level, and <level> is one of the valid logger levels (off, fatal, severe, warning, audit, info, config, detail, fine, finer, finest, all). Separate multiple log detail level specifications with colons (:).

Components correspond to Java packages and classes, or to collections of Java packages. Use an asterisk (*) as a wildcard to indicate components that include all the classes in all the packages that are contained by the specified component. For example:

*****
Specifies all traceable code running in the application server, including the product system code and customer code.

**com.ibm.ws.***
Specifies all classes with the package name beginning with com.ibm.ws.

**com.ibm.ws.classloader.JarClassLoader**
Specifies the JarClassLoader class only.

An error can occur when setting a log detail level specification from the administrative console if selections are made from both the Groups and Components lists. In some cases, the selection made from one list is lost when adding a selection from the other list. To work around this problem, enter the log detail level specification directly into the log detail level entry field.

Select a component or group to set a log detail level. The table following lists the valid levels for application servers at WebSphere Application Server Version 6 and later, and the valid logging and trace levels for earlier versions:

| Version 6 logging level | Logging level before Version 6 | Trace level before Version 6 | Content / Significance |
| --- | --- | --- | --- |
| Off | Off | All disabled* | Logging is turned off.<br><br>* In Version 6, a trace level of All disabled turns off trace, but does not turn off logging. Logging is enabled from the Info level. |
| Fatal | Fatal | - | Task cannot continue and component, application, and server cannot function. |
| Severe | Error | - | Task cannot continue but component, application, and server can still function. This level can also indicate an impending fatal error. |
| Warning | Warning | - | Potential error or impending error. This level can also indicate a progressive failure (for example, the potential leaking of resources). |
| Audit | Audit | - | Significant event affecting server state or resources |
| Info | Info | - | General information outlining overall task progress |
| Config | - | - | Configuration change or status |
| Detail | - | - | General information detailing subtask progress |

| | | | |
|---|---|---|---|
| Fine | - | Event | Trace information - General trace + method entry, exit, and return values |
| Finer | - | Entry/Exit | Trace information - Detailed trace |
| Finest | - | Debug | Trace information - A more detailed trace that includes all the detail that is needed to debug problems |
| All | | All enabled | All events are logged. If you create custom levels, All includes those levels, and can provide a more detailed trace than finest. |

When you enable a logging level in Version 6.0 or above, you are also enabling all of the levels with higher severity. For example, if you set the logging level to warning on your Version 6.x application server, then warning, severe and fatal events are processed.

Trace information, which are events at the Fine, Finer and Finest levels, can be written only to the trace log. Therefore, if you do not enable diagnostic trace, setting the log detail level to Fine, Finer, or Finest will not have an effect on the data that is logged.

## Loggers

Loggers are used by applications and runtime components to capture message and trace events.

When situations occur that are significant either due to a change in state, for example when a server completes startup or because a potential problem is detected, such as a timeout waiting for a resource, a message is written to the logs. Trace events are logged in debugging scenarios, where a developer needs a clear view of what is occurring in each component to understand what might be going wrong. Logged events are often the only events available when a problem is first detected, and are used during both problem recovery and problem resolution.

Loggers are organized hierarchically. Each logger can have zero or more child loggers.

Loggers can be associated with a resource bundle. If specified, the resource bundle is used by the logger to localize messages that are logged to the logger. If the resource bundle is not specified, a logger uses the same resource bundle as its parent.

You can configure loggers with a level. If specified, the level is compared by the logger to incoming events. The events that are less severe than the level set for the logger are ignored by the logger. If the level is not specified, a logger takes on the level that is used by its parent. The default level for loggers is Level.INFO.

Loggers can have zero or more attached handlers. If supplied, all events that are logged to the logger are passed to the attached handlers. Handlers write events to output destinations such as log files or network sockets. When a logger finishes passing a logged event to all of the handlers that are attached to that logger, the logger passes the event to the handlers that are attached to the parents of the logger. This process stops if a parent logger is configured not to use its parent handlers. Handlers in WebSphere Application Server are attached to the root logger. Set the useParentHandlers logger property to `false` to prevent the logger from writing events to handlers that are higher in the hierarchy.

Loggers can have a filter. If supplied, the filter is invoked for each incoming event to tell the logger whether or not to ignore it.

Applications interact directly with loggers to log events. To obtain or create a logger, a call is made to the Logger.getLogger method with a name for the logger. Typically, the logger name is either the package qualified class name or the name of the package that the logger is used by. The hierarchical logger namespace is automatically created by using the dots in the logger name. For example, the com.ibm.websphere.ras logger has a com.ibm.websphere parent logger, which has a com.ibm parent. The parent at the top of the hierarchy is referred to as the *root logger*. This root logger is created during initialization. The root logger is the parent of the com logger.

Loggers are structured in a hierarchy. Every logger, except the root logger, has one parent. Each logger can also have 0 or more children. A logger inherits log handlers, resource bundle names, and event filtering settings from its parent in the hierarchy. The logger hierarchy is managed by the LogManager function.

Loggers create log records. A log record is the container object for the data of an event. This object is used by filters, handlers, and formatters in the logging infrastructure.

The logger provides several sets of methods for generating log messages. Some log methods take only a level and enough information to construct a message. Other, more complex logp (log precise) methods support the caller in passing class name and method name attributes, in addition to the level and message information. The logrb (log with resource bundle) methods add the capability of specifying a resource bundle as well as the level, message information, class name, and method name. Using methods such as severe, warning, fine, finer, and finest you can log a message at a particular level. For more information on logging and how to use it in your applications read “Using loggers in an application” on page 302. For a complete list of methods, see the java.util.logging documentation at http://java.sun.com/javase/.

## Log handlers

Log handlers write log record objects to output devices like log files, sockets, and notification mechanisms.

Loggers can have zero or more attached handlers. All objects that are logged to the logger are passed to the attached handlers, if handlers are supplied.

You can configure handlers with a level. The handler compares the level that is specified in the logged object to the level that is specified for the handler. If the level of the logged object is less severe than the level set in the handler, the object is ignored by the handler. The default level for handlers is ALL.

Handlers can have a filter. If a filter is supplied, the filter is invoked for each incoming object to tell the handler whether or not to ignore it.

Handlers can have a formatter. If a formatter is supplied, the formatter controls how the logged objects are formatted. For example, the formatter can decide to first include the time stamp, followed by a string representation of the level, followed by the message that is included in the logged object. The handler writes this formatted representation to the output device. Read “Example: Creating custom formatters with java.util.logging” on page 312 for information on using a custom formatter in your applications.

Both loggers and handlers can have levels and filters, and a logged object must pass all of these elements to be output. For example, you can set the logger level to FINE, but if the handler level is set at WARNING, only WARNING level messages are displayed in the output for that handler. Conversely, if your log handler is set to output all messages (level=All), but the logger level is set to WARNING, the logger never sends messages lower than WARNING to the log handler.

## Log levels

Levels control which events are processed by Java logging. WebSphere Application Server controls the levels of all loggers in the system.

The level value is set from configuration data when the logger is created and can be changed at run time from the administrative console. If a level is not set in the configuration data, a level is obtained by proceeding up the hierarchy until a parent with a level value is found. You can also set a level for each handler to indicate which events are published to an output device. When you change the level for a logger in the administrative console, the change is propagated to the children of the logger.

Levels are cumulative; a logger can process logged objects at the level that is set for the logger, and at all levels above the set level. Valid levels are:

| Level | Content / Significance |
|---|---|
| Off | No events are logged. |
| Fatal | Task cannot continue and component cannot function. |
| Severe | Task cannot continue, but component can still function |
| Warning | Potential error or impending error |
| Audit | Significant event affecting server state or resources |
| Info | General information outlining overall task progress |
| Config | Configuration change or status |
| Detail | General information detailing subtask progress |
| Fine | Trace information - General trace + method entry / exit / return values |
| Finer | Trace information - Detailed trace |
| Finest | Trace information - A more detailed trace - Includes all the detail that is needed to debug problems |
| All | All events are logged. If you create custom levels, All includes your custom levels, and can provide a more detailed trace than Finest. |

For instructions on how to set logging levels, see “Configuring Java logging using the administrative console” on page 296

**Note:** Trace information, which includes events at the Fine, Finer and Finest levels, can be written only to the trace log. Therefore, if you do not enable diagnostic trace, setting the log detail level to Fine, Finer, or Finest does not effect the logged data.

## Log filters

Log filters help control more detailed logging settings that are not handled by usual log level settings.

A filter provides an optional, secondary control over what is logged, beyond the control that is provided by setting the level. Applications can apply a filter mechanism to control logging output through the logging APIs. An example of filter usage is to suppress all the events with a particular message key.

A filter is attached to a logger or log handler using the appropriate setFilter method. Read “Example: Creating custom filters with java.util.logging” on page 311 for information on implementing custom filters. For a complete list of filter methods, see the java.util.logging documentation at http://java.sun.com/javase/

## Log formatters

Log formatters format log messages so they can be used by various log handlers.

Handlers can be configured with a log formatter that knows how to format log records. The event, which is represented by the log record object, is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which writes the output to the output device.

The formatter is responsible for rendering the event for output. This formatter uses the resource bundle that is specified in the event to look up the message in the appropriate language.

Formatters are attached to handlers using the setFormatter method.

You can find the java.util.logging documentation at http://java.sun.com/javase/.

# Using loggers in an application

This topic describes how to use Java logging within an application.

## About this task

To create an application using Java logging, perform the following steps:

1. Create the necessary handler, formatter, and filter classes if you need your own log files.
2. If localized messages are used by the application, create a resource bundle, as described in “Creating log resource bundles and message files” on page 306.
3. In the application code, get a reference to a logger instance, as described in “Using a logger.”
4. Insert the appropriate message and trace logging statements in the application, as described in “Using a logger.”

## Using a logger

You can use Java logging to log messages and add tracing.

### About this task

Use WsLevel.DETAIL level and above for messages, and lower levels for trace. The WebSphere Application Server Extension API (the com.ibm.websphere.logging package) contains the WsLevel class.

For messages use:

```
WsLevel.FATAL
Level.SEVERE
Level.WARNING
WsLevel.AUDIT
Level.INFO
Level.CONFIG
WsLevel.DETAIL
```

For trace use:

```
Level.FINE
Level.FINER
Level.FINEST
```

1. Use the logp method instead of the log or the logrb method. The logp method accepts parameters for class name and method name. The log and logrb methods will generally try to infer this information, but the performance penalty is prohibitive. In general, the logp method has less performance impact than the log or the logrb method.
2. Avoid using the logrb method. This method leads to inefficient caching of resource bundles and poor performance.
3. Use the isLoggable method to avoid creating data for a logging call that does not get logged. For example:

```
if (logger.isLoggable(Level.FINEST)) {
  String s = dumpComponentState(); // some expensive to compute method
  logger.logp(Level.FINEST, className, methodName, "componentX state
dump:\n{0}", s);
 }
```

## Example

The following sample applies to localized messages:

```
// note - generally avoid use of FINE, FINER, FINEST levels for messages to be consistent with
// WebSphere Application Server

String componentName = "com.ibm.websphere.componentX";
String resourceBundleName = "com.ibm.websphere.componentX.Messages";
Logger logger = Logger.getLogger(componentName, resourceBundleName);

// "Convenience" methods - not generally recommended due to lack of class
/ method names
//   - cannot specify message substitution parameters
//   - cannot  specify class and method names
if (logger.isLoggable(Level.SEVERE))
 logger.severe("MSG_KEY_01");

if (logger.isLoggable(Level.WARNING))
 logger.warning("MSG_KEY_01");

if (logger.isLoggable(Level.INFO))
 logger.info("MSG_KEY_01");

if (logger.isLoggable(Level.CONFIG))
 logger.config("MSG_KEY_01");


// log methods are not generally used due to lack of class and method
names
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(WsLevel.FATAL))
 logger.log(WsLevel.FATAL, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.log(Level.SEVERE, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.log(Level.WARNING, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.log(WsLevel.AUDIT, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.log(Level.INFO, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.log(Level.CONFIG, "MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.log(WsLevel.DETAIL, "MSG_KEY_01", "parameter 1");


// logp methods are the way to log
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
if (logger.isLoggable(WsLevel.FATAL))
 logger.logp(WsLevel.FATAL, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.logp(Level.SEVERE, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.logp(Level.WARNING, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.logp(WsLevel.AUDIT, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.logp(Level.INFO, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.logp(Level.CONFIG, className, methodName, "MSG_KEY_01",
"parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.logp(WsLevel.DETAIL, className, methodName, "MSG_KEY_01",
"parameter 1");


// logrb methods are not generally used due to diminished performance
```

```
of switching resource bundles dynamically
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
String resourceBundleNameSpecial =
"com.ibm.websphere.componentX.MessagesSpecial";

if (logger.isLoggable(WsLevel.FATAL))
 logger.logrb(WsLevel.FATAL, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.SEVERE))
 logger.logrb(Level.SEVERE, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.WARNING))
 logger.logrb(Level.WARNING, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.AUDIT))
 logger.logrb(WsLevel.AUDIT, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.INFO))
 logger.logrb(Level.INFO, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(Level.CONFIG))
 logger.logrb(Level.CONFIG, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");

if (logger.isLoggable(WsLevel.DETAIL))
 logger.logrb(WsLevel.DETAIL, className, methodName, resourceBundleNameSpecial,
"MSG_KEY_01", "parameter 1");
```

For trace, or content that is not localized, the following sample applies:

```
// note - generally avoid use of FATAL, SEVERE, WARNING, AUDIT,
// INFO, CONFIG, DETAIL levels for trace
// to be consistent with WebSphere Application Server

String componentName = "com.ibm.websphere.componentX";
Logger logger = Logger.getLogger(componentName);

// Entering / Exiting methods are used for non trivial methods
if (logger.isLoggable(Level.FINER))
 logger.entering(className, methodName);

if (logger.isLoggable(Level.FINER))
 logger.entering(className, methodName, "method param1");

if (logger.isLoggable(Level.FINER))
 logger.exiting(className, methodName);

if (logger.isLoggable(Level.FINER))
 logger.exiting(className, methodName, "method result");


// Throwing method is not generally used due to lack of message - use
logp with a throwable parameter instead
if (logger.isLoggable(Level.FINER))
 logger.throwing(className, methodName, throwable);


// Convenience methods are not generally used due to lack of class
/ method names
//   - cannot specify message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(Level.FINE))
 logger.fine("This is my trace");

if (logger.isLoggable(Level.FINER))
 logger.finer("This is my trace");

if (logger.isLoggable(Level.FINEST))
 logger.finest("This is my trace");


// log methods are not generally used due to lack of class and
method names
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - cannot specify class and method names
if (logger.isLoggable(Level.FINE))
 logger.log(Level.FINE, "This is my trace", "parameter 1");

if (logger.isLoggable(Level.FINER))
 logger.log(Level.FINER, "This is my trace", "parameter 1");
```

```
if (logger.isLoggable(Level.FINEST))
 logger.log(Level.FINEST, "This is my trace", "parameter 1");


// logp methods are the recommended way to log
//   - enable use of WebSphere Application Server-specific levels
//   - enable use of message substitution parameters
//   - enable use of class and method names
if (logger.isLoggable(Level.FINE))
 logger.logp(Level.FINE, className, methodName, "This is my trace",
"parameter 1");

if (logger.isLoggable(Level.FINER))
 logger.logp(Level.FINER, className, methodName, "This is my trace",
"parameter 1");

if (logger.isLoggable(Level.FINEST))
 logger.logp(Level.FINEST, className, methodName, "This is my trace",
"parameter 1");


// logrb methods are not applicable for trace logging because no localization
is involved
```

## Configuring the logger hierarchy

WebSphere Application Server handlers are attached to the Java root logger, which is at the top of the logger hierarchy. As a result, any request from anywhere in the logger tree can be processed by WebSphere Application Server handlers.

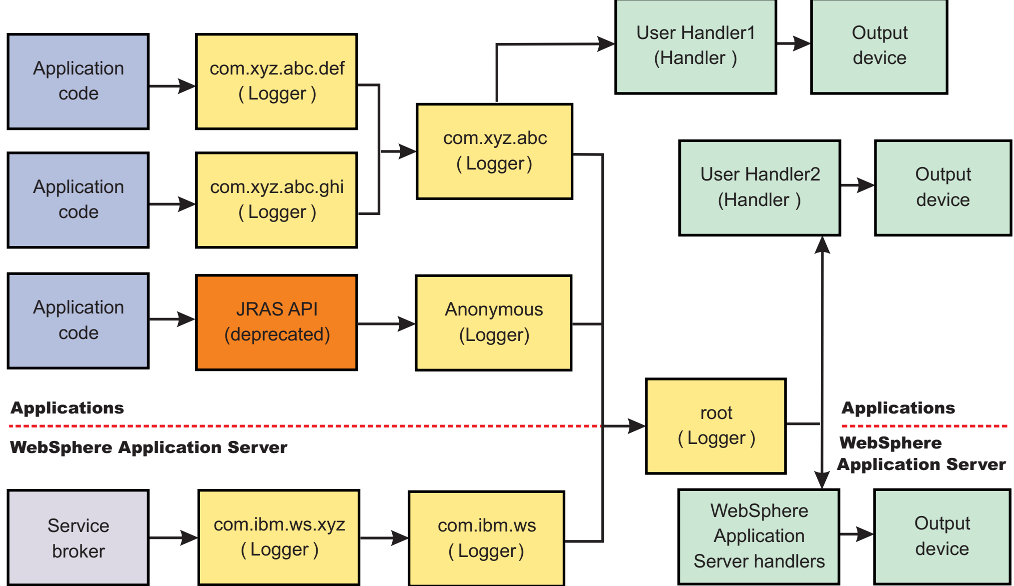
### About this task

You can configure your application server to handle logs in many different ways. Configure your log settings based upon your configuration and the logging structure that best suits your needs.

- Forward all application logging requests to the WebSphere Application Server handlers. This behavior is the default.
- Forward all application logging requests to your own custom handlers. Set the **useParentHandlers** option to `false` on one of your custom loggers, and then attach your handlers to that logger.
- Forward all application logging requests to both WebSphere Application Server handlers, and your custom handlers, but do not forward WebSphere Application Server logging requests to your custom handlers. Set the **useParentHandlers** option to `true` on one of your non-root custom loggers, and then attach your handlers to that logger.`True` is the default setting.
- Forward all WebSphere Application Server logging requests to both WebSphere Application Server handlers, and your custom handlers. Logging requests are always forwarded to WebSphere Application Server handlers. To forward WebSphere Application Server requests to your custom handlers, attach your custom handlers to the Java root logger, so that they are at the same level in the hierarchy as the WebSphere Application Server handlers.

### Example

The following example shows how these requirements can be met using the Java logging infrastructure:




## Creating log resource bundles and message files

You can forward messages that are written to the internal WebSphere Application Server logs to other processes for display. Messages that are displayed on the administrative console, which can be running in a different location than the server process, can be localized using the *late binding* process. Late binding means that WebSphere Application Server does not localize messages when they are logged, but defers localization to the process that displays the message.

### About this task

Every method that accepts messages localizes those messages. The mechanism for providing localized messages is the resource bundle support provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.ResourceBundle, java.util.ListResourceBundle and java.util.PropertyResourceBundle classes, as well as the java.text.MessageFormat class.

The PropertyResourceBundle class is the preferred mechanism to use.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. You must package the resource bundle separately from the application, and install it in a location where the viewing process can access it.

By default, the WebSphere Application Server runtime localizes all the messages when they are logged. This localization eliminates the need to pass a `.jar` file to the application, unless you need to localize in a different location. However, you can use the early binding technique to localize messages as they log. An application that uses early binding must localize the message before logging it. The application looks up the localized text in the resource bundle and formats the message. Use the early binding technique to package the application resource bundles with the application.

To create a resource bundle, perform the following steps.

1. Create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:
   - Each property in the file is terminated with a line-termination character.
   - If a line contains white space only, or if the first non-white space character of the line is the pound sign symbol (#) or exclamation mark (!), the line is ignored. The # and ! characters can therefore be used to put comments into the file.
   - Each line in the file, unless it is a comment or consists of white space only, denotes a single property. A backslash (\) is treated as the line-continuation character.
   - The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (=), colon (:), and white space ( ).
   - The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but doing this process is not recommended, because escaping characters is error prone and confusing. Instead, use a valid separator character that does not display in any keys in the properties file.
   - White space after the key and separator is ignored until the first non-white space character is encountered. All characters remaining before the line-termination character define the element.

   See the Java documentation for the java.util.Properties class for a full description of the syntax and the construction of properties files.
2. Translate the file into localized versions of the file with language-specific file names. For example, a file named `DefaultMessages.properties` can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese.
3. When the translated resource bundles are available, put the bundle in a directory that is part of the application class path.
4. When a message logger is obtained from the log manager, configure it to use a particular resource bundle. Messages logged with the Logger API use this resource bundle when message localization is performed. At run time, the user locale setting determines the properties file from which to extract the message that is specified by a message key, ensuring that the message is delivered in the correct language.
5. If the message loggers msg method is called, a resource bundle name must be explicitly provided.

## What to do next

The application locates the resource bundle based on the file location relative to any directory in the class path. For instance, if the DefaultMessages.properties property resource bundle is located in the *baseDir*/*subDir1*/*subDir2*/`resources` directory and *baseDir* is in the class path, the name *subdir1*.*subdir2*.`resources.DefaultMessage` is passed to the message logger to identify the resource bundle.

***Example: Logging resource bundles by creating a properties file:***

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a properties resource bundle. This sample shows how to create such a properties file.

**Resource bundle sample**

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs are inserted. All the normal properties file conventions and rules apply to this file. In addition, the creator must be aware of other restrictions that are imposed on the values by the Java MessageFormat class. For example, apostrophes must be escaped or they cause a problem. Avoid the use of non-portable characters. WebSphere Application Server does not support the use of extended formatting conventions that the MessageFormat class supports, such as {1, date} or {0,number, integer}.

Assume that the base directory for the application that uses this resource bundle is `baseDir` and that this directory is in the class path. Assume that the properties file is stored in the subdirectory `baseDir` that is

not in the class path (for example, `baseDir/subDir1/subDir2/resources`). To allow the messages file to resolve, the `subDir1.subDir2.resources.DefaultMessage` name is used to identify the property resource bundle and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`.

```
# Contents of the DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three parameter: parm1={0}, parm2 = {1}, parm3={2}
```

When the `DefaultMessages.properties` file is created, the file can be sent to a translation center where the localized versions are generated.

## Changing the message IDs used in log files

You can change the default format for message IDs in server logs by setting the com.ibm.websphere.logging.messageId.version system property.

### Before you begin

**Note:** Beginning with WebSphere Application Server Version 6.0, logging files are formatted according to a standardized system. However, the default runtime behavior is still configured to use the older format. In new releases of WebSphere Application Server, the message IDs that are written to log files will be changed to ensure they do not conflict with other IBM products. The default runtime behavior is still configured to use the older message IDs, deprecated in Version 7.0.

As a result of the default runtime behavior, you might see a mixture of messages that use 4–letter message prefixes and 5–letter message prefixes. The information in this topic explains how to change your configuration so that the messages consistently show with 5–letter message prefixes. The default behavior has not changed to minimize the impact on customers that depend on the existence of the 4–letter message prefixes.

The following is a sample of an entry in a trace.log file using a default message ID. Note that the message ID is PMON0001A

```
[1/26/05 10:17:12:529 EST] 0000000a PMIImpl       A   PMON0001A: PMI is enabled
```

A sample of the same entry using a new message ID follows. Note that the message ID is CWPMI0001A. All new WebSphere Application Server message IDs begin with 'CW'.

```
[1/26/05 10:17:12:529 EST] 0000000a PMIImpl       A   CWPMI0001A: PMI is enabled.
```

### About this task

If you are using a logging tool that uses the standardized format, you might want to change the default configuration settings to format the logging output appropriately. You will need to change the configuration for each Java virtual machine (JVM) in the cell if you want the output formatting to be the same across application servers.

- To configure logging files so that they use the newer, 5–letter error message prefixes for each process, use the following commands with the wsadmin utility:
  - Using Jacl:

```
$AdminConfig list JavaVirtualMachine
set cfgJvm [$AdminConfig list "JavaVirtualMachine"]
$AdminConfig create Property $cfgJvm {{name com.ibm.websphere.logging.messageId.version}
{value 6} {required false}}
$AdminConfig save
```

  - Using Jython:

```
ls = java.lang.System.getProperty("line.separator")
cfgJvmList = AdminConfig.list("JavaVirtualMachine").split(ls)
print cfgJvmList
```

```
cfgJvm = cfgJvmList[JavaVirtualMachine]
AdminConfig.create('Property', cfgJvm, [['name', 'com.ibm.websphere.logging.messageId.version'],
['value', '6'], ['required', 'false']]
AdminConfig.save()
```

Where *JavaVirtualMachine* is the number of the process that you want to use.

When you specify the process, the first process listed is zero (0), the second process is one (1), and so on. Make the changes for each JVM in the cell for consistent output formatting.

**Note:** Restart the application server for the changes to take effect.

- To change the configuration so that the log files contain the newer, 5–letter message prefixes in the startServer.log or stopServer.log files, modify the startServer and stopServer scripts in the `install_root/bin` directory. Within these files, add the following line of code:

```
>> %TMPJAVAPROPFILE% echo com.ibm.websphere.logging.messageId.version=6
```

**Note:** Restart the application server for the changes to take effect.

## Results

Message IDs written to log files will now be compliant with the new standard.

### *Converting log files to use IBM unique Message IDs:*

The convertlog command creates a new log file with either new or old message IDs substituted in place of the message IDs in the source file.

#### Before you begin

Prior to Version 6.x, components were assigned message IDs that are not necessarily unique across IBM software products. In Version 6.0, a system property was provided to map the message IDs in output logs to a set of IBM unique message IDs (all WebSphere Application Server message IDs now start with CW) that do not conflict with other IBM software products. The default runtime behavior still uses the old message IDs.

#### About this task

To facilitate the migration of logging tools that are reliant on the old message IDs, the convertlog command is provided to convert the message IDs of log entries from the old standard to the new standard, or the new standard back to the old. By default, the software is configured to use the old message IDs when logging, but you can change the default output with the com.ibm.websphere.logging.messageId.version system property. Read “Changing the message IDs used in log files” on page 308 for more information.

Use the convertlog command to convert the log output:

```
convertlog <source file name> <destination file name> [options]
       options: -newMessageFormat convert message IDs to CCCCCnnnnS format
                 (cannot be used with -m5)
                -oldMessageFormat convert message IDs to CCCCnnnnS format
                 (cannot be used with -m6)
```

#### Results

After using the convertlog command you have a new file with message IDs in the chosen format.

### *convertlog command:*

The convertlog command is used to convert the message IDs in log entries from the old standard to the new standard, or the new standard back to the old.

Previous versions of WebSphere Application Server used message IDs that are deprecated in WebSphere Application Server Version 7.0. To facilitate the migration of tools based on the old message IDs, the convertlog command is implemented to translate log files from one message ID standard to the other.

Use the convertlog command as follows:

```
convertlog <source file name> <destination file name> [options]
       options: -newMessageFormat convert message IDs to CCCCCnnnnS format
                 (cannot be used with -m5)
                -oldMessageFormat convert message IDs to CCCCnnnnS format
                 (cannot be used with -m6)
```

***MessageConverter class:***

The com.ibm.websphere.logging.MessageConverter class provides a method to convert a message ID at the front of a String into either a new message ID or an old message ID. The direction of the conversion is controlled with the *conversionType* argument.

Use the MessageConverter class with log analysis tools to convert message IDs from earlier versions of WebSphere Application Server into the corresponding message IDs that are used in later releases, or to revert message IDs to an earlier format.

**Method**

```
public static java.lang.String convert(java.lang.String in, short conversionType)
```

**Parameters**

Use the following parameters with the `MessageConverter` class:

| Parameter Name | Description |
|---|---|
| *in* | The message to convert. The method assumes the message ID is the first part of the supplied message with no leading white space. |
| *conversionType* | CONVERSION_TYPE_WASV5_TO_WASV6 |
| | CONVERSION_TYPE_WASV6_TO_WASV5 |

## Example: Creating custom log handlers with java.util.logging

There may be occasions when you want to propagate log records to your own log handlers rather than participate in integrated logging.

To use a stand-alone log handler, set the useParentHandlers flag to `false` in your application.

The mechanism for creating a customer handler is the Handler class support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with handlers, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.logging API.

The following sample shows a custom handler:

```
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.logging.Handler;
import java.util.logging.LogRecord;

/**
 * MyCustomHandler outputs contents to a specified file
 */
public class MyCustomHandler extends Handler {
```

```
 FileOutputStream fileOutputStream;
 PrintWriter printWriter;

 public MyCustomHandler(String filename) {
  super();

  // check input parameter
  if (filename == null || filename == "")
   filename = "mylogfile.txt";

  try {
   // initialize the file
   fileOutputStream = new FileOutputStream(filename);
   printWriter = new PrintWriter(fileOutputStream);
    setFormatter(new SimpleFormatter());
  }
  catch (Exception e) {
   // implement exception handling...
  }
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#publish(java.util.logging.LogRecord)
  */
 public void publish(LogRecord record) {
  // ensure that this log record should be logged by this Handler
  if (!isLoggable(record))
   return;

  // Output the formatted data to the file
  printWriter.println(getFormatter().format(record));
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#flush()
  */
 public void flush() {
  printWriter.flush();
 }

 /* (non-API documentation)
  * @see java.util.logging.Handler#close()
  */
 public void close() throws SecurityException {
  printWriter.close();
 }
}
```

## Example: Creating custom filters with java.util.logging

A custom filter provides optional, secondary control over what is logged, beyond the control that is provided by the level.

The mechanism for creating a customer filter is the Filter interface support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with filters, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation the for the java.util.logging API.

The following example shows a custom filter:

```
/**
 * This class filters out all log messages starting with SECJ022E, SECJ0373E, or SECJ0350E.
 */
import java.util.logging.Filter;
import java.util.logging.Handler;
import java.util.logging.Logger;
import java.util.logging.LogRecord;
```

```
public class MyFilter implements Filter {
 public boolean isLoggable(LogRecord lr) {
  String msg = lr.getMessage();
  if (msg.startsWith("SECJ0222E") || msg.startsWith("SECJ0373E") || msg.startsWith("SECJ0350E")) {
   return false;
  }
  return true;
 }
}

//This code will register the above log filter with the root Logger's handlers (including the WAS system logs):
...
Logger rootLogger = Logger.getLogger("");
rootLogger.setFilter(new MyFilter());
```

## Example: Creating custom formatters with java.util.logging

A formatter formats events. Handlers are associated with one or more formatters.

The mechanism for creating a customer formatter is the Formatter class support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with formatters, as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.logging API.

The following example shows a custom formatter:

```
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;

/**
 * MyCustomFormatter formats the LogRecord as follows:
 * date   level   localized message with parameters
 */
public class MyCustomFormatter extends Formatter {

 public MyCustomFormatter() {
  super();
 }

 public String format(LogRecord record) {

  // Create a StringBuffer to contain the formatted record
  // start with the date.
  StringBuffer sb = new StringBuffer();

  // Get the date from the LogRecord and add it to the buffer
  Date date = new Date(record.getMillis());
  sb.append(date.toString());
  sb.append(" ");

  // Get the level name and add it to the buffer
  sb.append(record.getLevel().getName());
  sb.append(" ");

  // Get the formatted message (includes localization
  // and substitution of paramters) and add it to the buffer
  sb.append(formatMessage(record));
  sb.append("\n");

  return sb.toString();
 }
}
```

## Example: Adding custom handlers, filters, and formatters

In some cases you might want to have your own custom log files. Adding custom handlers, filters, and formatters enables you to customize your logging environment beyond what can be achieved by the configuration of the default WebSphere Application Server logging infrastructure.

The following example demonstrates how to add a new handler to process requests to the com.myCompany subtree of loggers (see “Configuring the logger hierarchy” on page 305). The main method in this sample gives an example of how to use the newly configured logger.

```
import java.util.Vector;
import java.util.logging.Filter;
import java.util.logging.Formatter;
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyCustomLogging {

 public MyCustomLogging() {
  super();
 }

 public static void initializeLogging() {

  // Get the logger that you want to attach a custom Handler to
  String defaultResourceBundleName = "com.myCompany.Messages";
  Logger logger = Logger.getLogger("com.myCompany", defaultResourceBundleName);

  // Set up a custom Handler (see MyCustomHandler example)
  Handler handler = new MyCustomHandler("MyOutputFile.log");

  // Set up a custom Filter (see MyCustomFilter example)
  Vector acceptableLevels = new Vector();
  acceptableLevels.add(Level.INFO);
  acceptableLevels.add(Level.SEVERE);
  Filter filter = new MyCustomFilter(acceptableLevels);

  // Set up a custom Formatter (see MyCustomFormatter example)
  Formatter formatter = new MyCustomFormatter();

  // Connect the filter and formatter to the handler
  handler.setFilter(filter);
  handler.setFormatter(formatter);

  // Connect the handler to the logger
  logger.addHandler(handler);

  // avoid sending events logged to com.myCompany showing up in WebSphere
  // Application Server logs
  logger.setUseParentHandlers(false);

 }

 public static void main(String[] args) {
  initializeLogging();

  Logger logger = Logger.getLogger("com.myCompany");

  logger.info("This is a test INFO message");
  logger.warning("This is a test WARNING message");
  logger.logp(Level.SEVERE, "MyCustomLogging", "main", "This is a test SEVERE message");
 }
}
```

When the above program is run, the output of the program is written to the MyOutputFile.log file. The content of the log is in the expected log file, as controlled by the custom handler, and is formatted as defined by the custom formatter. The warning message is filtered out, as specified by the configuration of the custom filter. The output is as follows:

```
C:\>type MyOutputFile.log
Sat Sep 04 11:21:19 EDT 2004 INFO This is a test INFO message
Sat Sep 04 11:21:19 EDT 2004 SEVERE This is a test SEVERE message
```

# HTTP error, FRCA, and NCSA access log settings

Use this page to configure the global HTTP error log, and National Center for Supercomputing Applications (NCSA) access log settings for an HTTP inbound channel. If you are running the product on z/OS, you can also use this page to configure the global Fast Response Cache Accelerator (FRCA) log settings for an HTTP inbound channel. FRCA logs are a specialized form of NCSA logs and can only be created in a z/OS environment.

To view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* **> HTTP error, NCSA access and FRCA logging**. This console page has separate sections for each type of logging. The FRCA logging section only appears if you are running the product on z/OS.

The HTTP error log contains a record of HTTP processing errors that occur. The level of error logging that occurs is dependent on the value that is selected for the Error log level field.

The NCSA access log contains a record of all inbound client requests that the HTTP transport channel handles. All of the messages that are contained in these logs are in NCSA format.

After you configure the HTTP error log, NCSA access logs, and FRCA logs, you must explicitly enable each type of logging on the settings page for the HTTP channels for which you want a specific types of logging to occur. To view the settings page for an HTTP channel, click **Servers > Server Types > Application servers** > *server* > **Web Container Settings** > **Web container transport chains** > **HTTP inbound channel**.

**Note:** The settings for any of these logs can also be modified on the settings page for a specific HTTP inbound channel. Any changes that you make on the HTTP inbound channel settings page only apply to that specific inbound channel. and override any global configuration settings that you specify on this page.

## Enable logging service at server start-up

Select this option if you want any of the following logging to start when the server starts:

- NCSA access logging
- HTTP error logging

**Note:** Even if you select this option, you must explicitly enable the type of logging that you want to occur on this page and on the settings page for the HTTP transport channel for which you want that type of logging to occur.

## Enable NCSA access logging

When selected, a record of inbound client requests that the HTTP transport channel handles is kept in the NCSA access log.

## NCSA access log file path

Specifies the directory path and name of the NCSA access log. Standard variable substitutions, such as $(SERVER_LOG_ROOT), can be used when specifying the directory path.

## NCSA access log maximum size

Specifies the maximum size, in megabytes, of the NCSA access log. When this size is reached, the *logfile_name*.1 archive log is created. However, every time that the original log file overflows this archive file, the file is overwritten with the most current version of the original log file.

## Maximum number of historical files

Specifies the maximum number of historical versions of the NCSA access log file that are kept for future reference.

### NCSA access log format

Specifies which NCSA format is used when logging client access information. If you select Common, the log entries contain the requested resource and a few other pieces of information, but does not contain referral, user agent, and cookie information. If you select Combined, referral, user agent, and cookie information is included.

### Enable error logging

When selected, HTTP errors that occur while the HTTP channel processes client requests are recorded in the HTTP error log.

### Error log file path

Specifies the directory path and the name of the HTTP error log. Standard variable substitutions, such as $(SERVER_LOG_ROOT), can be used when specifying the directory path.

### Error log maximum size

Specifies the maximum size, in megabytes, of the HTTP error log file. When this size is reached, the *logfile_name*.1 archive log is created. However, every time that the original log file overflows this archive file, this file is overwritten with the most current version of the original log file.

### Maximum number of historical files

Specifies the maximum number of historical versions of the Error log file that are kept for future reference.

### Error log level

Specifies the type of error messages that are included in the HTTP error log.

You can select:

**Critical**
Only critical failures that stop the Application Server from functioning properly are logged.

**Error**
The errors that occur in response to clients are logged. These errors require Application Server administrator intervention if they result from server configuration settings.

**Warning**
Information on general errors, such as socket exceptions that occur while handling client requests, are logged. These errors do not typically require Application Server administrator intervention.

**Information**
The status of the various tasks that are performed while handling client requests is logged.

**Debug**
More verbose task status information is logged. This level of logging is not intended to replace RAS logging for debugging problems, but does provide a steady status report on the progress of individual client requests. If this level of logging is selected, you must specify a large enough log file size in the Error log maximum size field to contain all of the information that is logged.

## Logger.properties file for configuring logger settings

Use the Logger.properties file to set logger attributes for specific loggers.

The properties file is loaded the first time that the Logger.getLogger(*logger_name*) method is called within an application.

**Important:** The name of the Logger.properties file is case sensitive. Use a capital ″L″ in the file name.

When an application calls the Logger.getLogger method for the first time, all the available logger properties files are loaded. Applications can provide Logger.properties files in:

- the META-INF directory of the Java archive (JAR) file for the application
- directories included in the class path of an application module

- directories included in the application class path

The properties file contains two categories of parameters, logger control and logger data:

- Logger control information
  - Minimum localization level: The minimum LogRecord level for which localization is attempted
  - Group: The logical group that this component belongs to
  - Event factory: The Common Base Event template file to use with the event factory. The naming convention for this template is the fully qualified component name, with a file extension of `.event.xml`. For example, a template that applies to the com.ibm.compXYZ package is called `com.ibm.compXYZ.event.xml`.
- Logger data information
  - Product name
  - Organization name
  - Component name
  - Extensions and additional properties

### Syntax of the Logger.properties file

Use the following syntax to set logger properties:

```
<logger base name>.<property>=value
```

where:

*logger base name* is the starting part of the logger name to which the property applies. All loggers with names starting with this string have the property applied.

*property* is one of the following properties:

- organization
- product
- component
- minimum_localization_level
- group
- eventfactory

### Sample Logger.properties file

In the following sample, the com.ibm.xyz.MyEventFactory event factory is used by any loggers in the com.ibm.websphere.abc package or any sub packages that do not override this value in their configuration file.

```
com.ibm.websphere.abc.eventfactory=com.ibm.xyz.MyEventFactory
```

### Group Logger.properties file

In the following example, the group is MyTraceGroup and the components are com.ibm.stuff and com.ibm.morestuff:

```
com.ibm.stuff.group=MyTraceGroup
com.ibm.morestuff.group=MyTraceGroup
```

## Example: Sample security policy for logging

Set up a security policy to allow your applications to modify logging and handler properties.

The sample security policy that follows grants access to the file system and runtime classes. Include this security policy, with the entry `permission java.util.logging.LoggingPermission "control"`, in the META-INF directory of your application if you want your applications to programmatically alter controlled properties of loggers and handlers. The META-INF file is located in the following locations for the different module types:

| | |
|---|---|
| EJB projects | ejbModule/META-INF/MANIFEST.MF |
| Application client projects | appClientModule/META-INF/MANIFEST.MF |
| Dynamic Web projects | WebContent/META-INF/MANIFEST.MF |
| Connector projects | connectorModule/META-INF/MANIFEST.MF |

Below is a sample security policy that grants permission to modify logging properties:

```
//////////////////////////////////////////////////
//
// WebSphere Application Server Security Policy
//
//////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Allow all access to the file system and runtime classes
////////////////////////////////////////////////////////////////////////
grant codeBase "file:${application}" {
   permission java.util.logging.LoggingPermission "control";
};
```

# Configuring applications to use Jakarta Commons Logging

Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging systems. WebSphere Application Server supports Jakarta Commons Logging by providing a logger. The support does not change interfaces defined by Jakarta Commons Logging.

## Before you begin

The WebSphere Application Server logger is a thin wrapper for the WebSphere Application Server logging facility. The logger name is *com.ibm.websphere.commons.logging.WsJDK14Logger*. The logger can handle logging objects defined by either of the following:

- Java Logging found in Java Specification Request 47: Logging API Specification
- Common Base Event

A *logging object* is an object that holds logging entry information.

To better understand Jakarta Commons Logging, read Jakarta Commons and the specifications for Java Logging and for Common Base Event. To better understand use of the WebSphere Application Server logger, read “Jakarta Commons Logging” on page 318.

## About this task

WebSphere Application Server provides the Jakarta Commons Logging binary distribution in its `libraries` directory. By default, the product uses the Jakarta Commons Logging LogFactory implementation and JDK14Logger.

**Note:** The default configuration of Jakarta Commons Logging is stored in the commons-logging.properties file. To specify the factory class to use with Jakarta Commons Logging in an application, provide a file named org.apache.commons.logging.LogFactory, located in META-INF/services directory, that

contains the name of the factory class on the first line. This is the configuration mechanism for the JAR file service provider, as defined in JDK 1.3 and above.

For an application to use the WebSphere Application Server logger, the application must provide its own configuration for the logger. To configure an application to use the WebSphere Application Server logger, complete the steps that follow.

1. Examine “Configurations for the WebSphere Application Server logger” on page 320 and determine which configuration best suits your application.
2. Change your application configuration as needed to enable use of the WebSphere Application Server logger.

### Results

After the application starts, Jakarta Commons Logging routes the application’s logging output to the WebSphere Application Server logger.

## Jakarta Commons Logging

Jakarta Commons Logging provides a simple logging interface and thin wrappers for several logging systems. The logging interface enables application logging to be simple and independent of the logging system that the application uses. You can change the logging implementation for a deployed application without having to change the application logging code. However, the simplicity of the logging interface prevents the application from leveraging all the functionality of the logging systems.

This topic provides the following information about Jakarta Commons Logging in WebSphere Application Server:

- “Support for Jakarta Commons Logging”
- “Benefits of support for Jakarta Commons Logging”
- “Overview of the process for using Jakarta Commons Logging” on page 319
- “Classes used to obtain a logger factory and logger” on page 319
- “Logger level configuration and mapping” on page 320

### Support for Jakarta Commons Logging

The product supports Jakarta Commons Logging by providing a logger, a thin wrapper for the WebSphere Application Server logging facility. The logger can handle both Java Logging (JSR-47) and Common Base Event logging objects. A *logging object* is an object that holds logging entry information.

The product support for Jakarta Commons Logging does not change interfaces defined by Jakarta Commons Logging.

### Benefits of support for Jakarta Commons Logging

The WebSphere Application Server support for Jakarta Commons Logging provides the following benefits:

- WebSphere Application Server is pre-configured to use Jakarta Commons Logging.

  All of the functionality of Jakarta Commons Logging is provided for any application or WebSphere Application Server component. Logging calls are routed by default to the underlying WebSphere Application Server logging facility.
- A logger that uses the WebSphere Application Server logging facility.

  Applications and components can pass both Java Logging and Common Base Event logging objects to the WebSphere Application Server logger without conversion to strings, providing applications with enhanced logging. Further, Jakarta Commons Logging Logger levels are integrated into WebSphere Application Server administrative facilities.

## Overview of the process for using Jakarta Commons Logging

Logging with Jakarta Commons Logging consists of the steps that follow. "Configurations for the WebSphere Application Server logger" on page 320 provides details on configuring your application to use the WebSphere Application Server logger.

1. Obtain an instance of a logger factory.

   To obtain a logger factory, use Jakarta Commons Logging code. You can configure the code to meet your needs. In WebSphere Application Server, Jakarta Commons Logging is configured by default to instantiate the Jakarta Commons Logging default logger factory. Applications or WebSphere Application Server components can provide their own configuration if they use a different logger factory implementation. Applications can use more than one factory.

2. Obtain an instance of a logger.

   To obtain a logger, use code implemented by a logger factory. Configuration of the code is implementation specific.

   The WebSphere Application Server logger implements the methods defined in the logging interface. The logging methods take at least one argument, which can be any Java object. The WebSphere Application Server logger, the WsJDK14Logger logger described in "Classes used to obtain a logger factory and logger," handles the following objects passed into the following logging methods:

   **CommonBaseEvent**
   Wrapped into CommonBaseEventLogRecord

   **CommonBaseEventLogRecord**
   Passed without change

   **LogRecord**
   Passed without change

   **Other objects**
   Converted to String

   Applications or WebSphere Application Server components can provide their own configuration if they use an implementation of a logger that is not specific to WebSphere Application Server. An application must know what factory is being used in order to configure it.

3. Start your application. Jakarta Commons Logging routes the application's logging output to the designated logger

## Classes used to obtain a logger factory and logger

| Class name | Description |
|---|---|
| LogFactory | *LogFactory* is a Jakarta Commons Logging class that implements initialization logic. LogFactory is an abstract class that every logger factory implementation has to extend. It provides static methods for obtaining:<br>• An instance of a factory class<br>• Instances of a logger, using an instance of the factory class<br><br>LogFactory provides methods for obtaining instances of loggers, although these methods delegate the logger instantiation and configuration to an instance of a logger factory class.<br><br>Logger factories, once instantiated, are cached on a per context class loader basis. The instances in a cache can be released. This functionality is designed for platform container implementations rather than for applications. |
| LogFactoryImpl | *LogFactoryImpl* is a Jakarta Commons Logging concrete class that implements the default logger factory using methods in LogFactory. To use Java Logging, there must always be at least one instance of a logger factory class, even if the application has not explicitly obtained one. If the configuration does not name a logger factory class, LogFactoryImpl is used as the default. |

| Class name | Description |
| --- | --- |
| Log | *Log* is a Jakarta Commons Logging interface for loggers. Commons logging loggers have to implement the Log interface. Because the goal of Jakarta Commons Logging is to wrapper any logging system, the Log interface defines a small set of common logging methods. In WebSphere Application Server, WsJDK14Logger implements the Log interface.<br><br>Logger instantiation and configuration is specific to every logger factory. Logging in WebSphere Application Server uses the default logger factory provided in Jakarta Commons Logging, which keeps instantiated loggers in cache, on a per class loader context basis. |
| WsJDK14Logger | *WsJDK14Logger* is a WebSphere Application Server class that provides a Jakarta Commons Logging logger by implementing the Log interface. The WsJDK14Logger logger differs from the Java Logging logger in that the WsJDK14Logger logger enables Java Logging or Common Base Event objects to be passed over without converting them into String objects. This prevents any information loss the conversion to String might cause as well as allows the logging output to be more descriptive and precise. In contrast, the Java Logginglogger that is provided in Jakarta Commons Logging converts objects passed into the logging calls to String objects before passing them over to the underlying Java Logging. |

## Logger level configuration and mapping

Because Jakarta Commons Logging loggers are thin wrappers for specific logging systems, the loggers do not have their own level, but use the level of the logger from the underlying logging system. Although the underlying system can provide methods for changing level, there are no methods for changing level defined on the Log interface, which all Jakarta Commons Logging logger must implement. WsJDK14Logger uses the level of its underlying Java Logging logger.

Following table shows, on the left, the mapping of Jakarta Commons Logging levels within WsJDK14Logger to levels in the WebSphere Application Server implementation of Java Logging. On the right, it shows the levels defined in Java Logging and the level mapping in the Jakarta Commons Logging JDK14Logger to the Java Logging levels.

| WsJDK14Logger | Java Logging in WebSphere Application Server | Java Logging | JDK14Logger |
| --- | --- | --- | --- |
| Fatal | Fatal | | |
| Error | Severe | Severe | Fatal, Error |
| Warning | Warning | Warning | Warning |
| | Audit | | |
| Info | Info | Info | Info |
| | Config | Config | |
| | Detail | | |
| Debug | Fine | Fine | Debug |
| | Finer | Finer | |
| Trace | Finest | Finest | Trace |

The WsJDK14Logger level is synchronized with the underlying Java Logging logger level. WebSphere Application Server administration controls the WsJDK14Logger level.

# Configurations for the WebSphere Application Server logger

This topic describes several ways to configure an application to use the WebSphere Application Server logger.

The type of configuration that best suits an application depends upon the following:

- Whether the class loader order setting for the application is `Classes loaded with parent class loader first` (`Parent First`) or `Classes loaded with application class loader first` (`Parent Last`), you can set the class loader delegation mode on a console page. For more details about class load order and delegation, consult the class loading chapter in the *Developing and deploying applications* PDF book
- Whether Jakarta Commons Logging is bundled with the application configuration
- Whether Jakarta Commons Logging is provided within the application

The following tables describe the conditions required to enable an application to use the WebSphere Application Server logger.

**Class loader mode is `Parent First` and Jakarta Commons Logging is bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is not read** by the LogFactory because the parent class loader finds the WebSphere properties file first.<br><br>• The class name **is read** from the file `META-INF/services/org.apache.commons.logging.LogFactory` | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration, unless the configuration is provided in a META-INF file of the application or module. | The log used is either of the following:<br><br>• The Log implementation specified in the WebSphere Application Server default configuration<br><br>• An application-specific Log implementation if an application-specific LogFactory that instantiates a different Log implementation is used. | The **application parent class loader** is the first class loader to load the Jakarta Commons Logging code. The **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**. |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere default configuration. | The log used is the Log implementation specified in the WebSphere default configuration. | The Jakarta Commons Logging bundled with the application is not used. |

**Class loader mode is `Parent First` and Jakarta Commons Logging is not bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is not read** by the LogFactory because the parent class loader finds the WebSphere Application Server properties file first.<br><br>• The class name **is read** from the file `META-INF/services/ org.apache.commons .logging.LogFactory` | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration, unless the configuration is provided in a META-INF file of the application or module. | The log used is either of the following:<br><br>• The Log implementation specified in the WebSphere Application Server default configuration<br><br>• An application-specific Log implementation if an application-specific LogFactory that instantiates a different Log implementation is used. | The **application parent class loader** is the first class loader to load the Jakarta Commons Logging code. The **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**. |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration. | The log used is the Log implementation specified in the WebSphere Application Server default configuration. | Same as in the previous row |

### Class loader mode is `Parent Last` and Jakarta Commons Logging is bundled with the application

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
|---|---|---|---|
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is read** by the LogFactory because the class loader finds the application properties file first.<br><br>• The class name **is read** from the file `META-INF/services/ org.apache.commons .logging.LogFactory` | The log factory used is either of the following:<br><br>• The default Jakarta Commons Logging LogFactory<br><br>• The LogFactory specified in the application configuration | The log used is the Log implementation specified in the application configuration.<br><br>If the log factory used is the default Jakarta Commons Logging LogFactory, the Log implementation must be on the classpath of the application class loader. | The **application class loader** is the first class loader to load the Jakarta Commons Logging code. The **application bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **application class loader**. |

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
| --- | --- | --- | --- |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration. | The log used is the Log implementation specified in the WebSphere Application Server default configuration. | |

**Class loader mode is `Parent Last` and Jakarta Commons Logging is not bundled with the application**

| Jakarta Commons Logging configuration | LogFactory instance | Log instance | Comments |
| --- | --- | --- | --- |
| The application provides the configuration by either of the following:<br><br>• The properties file `commons-logging.properties` in the application classpath **is read** by the LogFactory because the class loader finds the application properties file first.<br><br>• The class name **is read** from the file `META-INF/services/org.apache.commons.logging.LogFactory` | The log factory used is either of the following:<br><br>• The default Jakarta Commons Logging LogFactory<br><br>• The LogFactory specified in the application configuration | The log used is the Log implementation specified in the application configuration.<br><br>If the log factory used is the default Jakarta Commons Logging LogFactory, the Log implementation must be on the classpath of the application class loader. | There is no Jakarta Commons Logging code at the application class loader. Thus, the **WebSphere bundle** that supports Jakarta Commons Logging provides the LogFactory static code that looks up the LogFactory configuration attributes.<br><br>For the static LogFactory code to instantiate the LogFactory instance specified in the application configuration, the LogFactory instance must be on the classpath of the **parent class loader**. |
| Not provided by the application | The log factory used is the LogFactory implementation specified in the WebSphere Application Server default configuration. | The log used is the Log implementation specified in the WebSphere Application Server default configuration. | |

# Programming with the JRas framework

Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server applications.

## Before you begin

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

## About this task

The JRas extensions allow message logging and diagnostic trace to work with WebSphere Application Server applications. They are based on the stand-alone JRas logging toolkit.

1. Retrieve a reference to the JRas manager.
2. Retrieve message and trace loggers by using methods on the returned manager.
3. Call the appropriate methods on the returned message and trace loggers to create message and trace entries, as appropriate.

# JRas logging toolkit

The JRas logging toolkit provides diagnostic information to help the administrator diagnose problems or tune application performance.

**Deprecated:** The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Developing, deploying, and maintaining applications are complex tasks. For example, when a running application encounters an unexpected condition, it might not be able to complete a requested operation. In such a case, you might want the application to inform the administrator that the operation failed and provide information. This action enables the administrator to take the proper corrective action. Those who develop or maintain applications might need to gather detailed information relating to the path of a running application to determine the root cause of a failure that is due to a code bug. The facilities that are used for these purposes are typically referred to as *message logging* and *diagnostic trace*.

Message logging (messages) and diagnostic trace (trace) are conceptually quite similar, but do have important differences. It is important for application developers to understand these differences to use these tools properly. To start with, the following operational definitions of messages and trace are provided.

**Message**
: A message entry is an informational record that is intended for end users, systems administrators and support personnel to view. The text of the message must be clear, concise, and interpretable. Messages are typically localized, meaning that they display in the national language of the end user. Although the destination and lifetime of messages might be configurable, some level of message logging is always enabled in normal system operation. Message logging must be used judiciously due to both performance considerations and the size of the message repository.

**Trace**
: A trace entry is an information record that is intended for service engineers or developers to use. This trace record might be considerably more complex, verbose, and detailed than a message entry. Localization support is typically not used for trace entries. Trace entries can be fairly inscrutable, understandable only by the appropriate developer or service personnel. It is assumed that trace entries are not written during normal runtime operation, but might be enabled as needed to gather diagnostic information.

WebSphere Application Server provides a message logging and diagnostic trace API that applications can use. This API is based on the stand-alone JRas logging toolkit, which was developed by IBM. The stand-alone JRas logging toolkit is a collection of interfaces and classes that provide message logging and diagnostic trace primitives. These primitives are not tied to any particular product or platform. The stand-alone JRas logging toolkit provides a limited amount of support, which is typically referred to as *systems management support*, including log file configuration support based on property files.

As designed, the stand-alone JRas logging toolkit does not contain the support that is required for integration into the WebSphere Application Server run time or for use in a Java 2 Platform, Enterprise Edition (J2EE) environment. To overcome these limitations, WebSphere Application Server provides a set of extension classes to address these shortcomings. This collection of extension classes is referred to as the JRas extensions. The JRas extensions do not modify the interfaces that are introduced by the

stand-alone JRas logging toolkit, but provide the appropriate implementation classes. The conceptual structure that is introduced by the stand-alone JRas logging toolkit is described in the following section. It is equally applicable to the JRas extensions.

## JRas concepts

The section contains a basic overview of important concepts and constructs that are introduced by the stand-alone JRas logging toolkit. This information is not an exhaustive overview of the capabilities of this logging toolkit, nor is it intended as a detailed discussion of usage or programming paradigms. More detailed information, including code examples, is available in JRas extensions and its subtopics, including in the API documentation for the various interfaces and classes that make up the logging toolkit.

**Event types**
The stand-alone JRas logging toolkit defines a set of event types for messages and a set of event types for trace. Examples of message types include informational, warning, and error. Examples of trace types include entry, exit, and trace.

**Event classes**
The stand-alone JRas logging toolkit defines both message and trace event classes.

**Loggers**
A logger is the primary object with which the user code interacts. Two types of loggers are defined: message loggers and trace loggers. The set of methods on message loggers and trace loggers are different because they provide different functionality. Message loggers create message records only and trace loggers create trace records only. Both types of loggers contain masks that indicate which categories of events the logger processes and which to ignore. Although every JRas logger is defined to contain both a message and trace mask, the message logger uses only the message mask and the trace logger uses the trace mask only. For example, by setting a message logger message mask to the appropriate state, it can be configured to process only error messages and ignore informational and warning messages. Changing the trace mask state of a message logger has no effect.

A logger contains one or more handlers to which it forwards events for further processing. When the user calls a method on the logger, the logger compares the event type that is specified by the caller to its current mask value. If the specified type passes the mask check, the logger creates an event object to capture the information relating to the event that passed to the logger method. This information can include information, such as the names of the class and method which logs the event, a message, and parameters to log, among others. When the logger creates the event object, it forwards the event to all handlers currently registered with the logger.

Methods that are used within the logging infrastructure do not make calls to the logger method. When an application uses an object that extends a thread class, implements the hashCode method, and makes a call to the logging infrastructure from that method, the result is a recursive loop.

**Handlers**
A handler provides an abstraction over an output device or event consumer. An example is a file handler, which knows how to write an event to a file. The handler also contains a mask that is used to further restrict the categories of events the handler processes. For example, a message logger might be configured to pass both warning and error events, but a handler attached to the message logger might be configured to pass error events only. Handlers also include formatters, which the handler invokes to format the data in the passed event before it is written to the output device.

**Formatters**
Handlers are configured with formatters, which know how to format events of certain types. A handler can contain multiple formatters, each of which knows how to format a specific class of event. The event object is passed to the appropriate formatter by the handler. The formatter returns formatted output to the handler, which then writes it to the output device.

# JRas Extensions

*JRas extensions* are the collection of implementation classes that support JRas integration into the WebSphere Application Server environment.

## JRas extensions

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The stand-alone JRas logging toolkit defines interfaces and provides a variety of concrete classes that implement these interfaces. Because the stand-alone JRas logging toolkit is developed as a general purpose toolkit, the implementation classes do not contain the configuration interfaces and methods that are necessary for use in the WebSphere Application Server product. In addition, many of the implementation classes are not written appropriately for use in a Java 2 Platform, Enterprise Edition (J2EE) environment. To overcome these shortcomings, WebSphere Application Server provides the appropriate implementation classes that support integration into the WebSphere Application Server environment. The collection of these implementation classes is referred to as the *JRas extensions*.

## Usage model

You can use the JRas extensions in three distinct operational modes:

**Integrated**
: In this mode, message and trace records are written only to logs that are defined and maintained by the WebSphere Application Server run time. This mode is the default mode of operation and is equivalent to the WebSphere Application Server V4.0 mode of operation.

**stand-alone**
: In this mode, message and trace records are written solely to stand-alone logs that are defined and maintained by the user. You control which categories of events are written to which logs, and the format in which entries are written. You are responsible for configuration and maintenance of the logs. Message and trace entries are not written to WebSphere Application Server runtime logs.

**Combined**
: In this mode, message and trace records are written to both WebSphere Application Server runtime logs and to stand-alone logs that you must define, control, and maintain. You can use filtering controls to determine which categories of messages and trace are written to which logs.

The JRas extensions are specifically targeted to an integrated mode of operation. The integrated mode of operation can be appropriate for some usage scenarios, but many scenarios are not adequately addressed by these extensions. Many usage scenarios require a stand-alone or combined mode of operation instead. A set of user extension points are defined that support JRas extensions in either a stand-alone or combined mode of operations.

## JRas extension classes

WebSphere Application Server provides a base set of implementation classes that are collectively referred to as the *JRas extensions*. Many of these classes provide the appropriate implementations of loggers, handlers, and formatters for use in a WebSphere Application Server environment.

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The collection of JRas classes is targeted at an integrated mode of operation. If you choose to use the JRas extensions in either stand-alone or combined mode, you can reuse the logger and manager class that are provided by the extensions, but you must provide your own implementations of handlers and formatters.

## WebSphere Application Server message and trace loggers

The message and trace loggers that are provided by the stand-alone JRas logging toolkit cannot be directly used in the WebSphere Application Server environment. The JRas extensions provide the appropriate logger implementation classes. Instances of these message and trace logger classes are obtained directly and exclusively from the WebSphere Application Server Manager class. You cannot directly instantiate message and trace loggers. Obtaining loggers in any manner other than directly from the Manager class is not allowed and directly violates the programming model.

The message and trace logger instances that are obtained from the WebSphere Application Server Manager class are subclasses of the RASMessageLogger and RASTraceLogger classes that are provided by the stand-alone JRas logging toolkit. The RASMessageLogger and RASTraceLogger classes define the set of methods that are directly available. Public methods that are introduced by the JRas extensions logger subclasses cannot be called directly by user code because it is a violation of the programming model.

Loggers are named objects and are identified by name. When the Manager class is called to obtain a logger, the caller is required to specify a name for the logger. The Manager class maintains a name-to-logger instance mapping. Only one instance of a named logger is ever created within the lifetime of a process. The first call to the Manager class with a particular name results in the logger, which is configured by the Manager class. The Manager class caches a reference to the instance, then returns it to the caller. Subsequent calls to the Manager class that specify the same name result in a returned reference to the cached logger. Separate namespaces are maintained for message and trace loggers. You can use a single name obtain both a message logger and a trace logger from the Manager, without ambiguity, and without causing a namespace collision.

In general, loggers have no predefined granularity or scope. A single logger can be used to instrument an entire application. You might determine that having a logger per class is more effective, or the appropriate granularity might be somewhere in between. Partitioning an application into logging domains is determined by the application writer.

The WebSphere Application Server logger classes that are obtained from the Manager class are thread-safe. Although the loggers provided as part of the stand-alone JRas logging toolkit implement the serializable interface, loggers are not serializable. Loggers are stateful objects, tied to a Java virtual machine instance and are not serializable. Attempting to serialize a logger is a violation of the programming model.

Personal or individual logger subclasses are not supported in a WebSphere Application Server environment.

## WebSphere Application Server handlers

WebSphere Application Server provides the appropriate handler class that is used to write message and trace events to the WebSphere Application Server run time logs. You cannot configure the WebSphere Application Server handler to write to any other destination. The creation of a WebSphere Application Server handler is a restricted operation and is not available to user code. Every logger that is obtained from the Manager comes preconfigured with an instance of this handler already installed. You can remove the WebSphere Application Server handler from a logger when you want to run in stand-alone mode. When you remove it, you cannot add the WebSphere Application Server handler again to the logger from which it is removed or any other logger. Also, you cannot directly call any method on the WebSphere Application Server handler. Attempting to create an instance of the WebSphere Application Server handler, to call methods on the WebSphere Application Server handler or to add a WebSphere Application Server handler to a logger by user code is a violation of the programming model.

### WebSphere Application Server formatters

The WebSphere Application Server handler comes preconfigured with the appropriate formatter for data that is written to WebSphere Application Server logs. The creation of a WebSphere Application Server formatter is a restricted operation and not available to user code. No mechanism exists that allows the user to obtain a reference to a formatter installed in a WebSphere Application Server handler, or to change the formatter a WebSphere Application Server handler is configured to use.

### WebSphere Application Server manager

WebSphere Application Server provides a Manager class in the com.ibm.websphere.ras package. All message and trace loggers must be obtained from this Manager class. A reference to the Manager class is obtained by calling the static Manager.getManager method. Message loggers are obtained by calling the createRASMessageLogger method on the Manager class. Trace loggers are obtained by calling the createRASTraceLogger method on the Manager class.

The manager also supports a *group* abstraction that is useful when dealing with trace loggers. The group abstraction supports multiple, unrelated trace loggers to register as part of a named entity called a *group*. WebSphere Application Server provides the appropriate systems management facilities to manipulate the trace setting of a group, similar to the way the trace settings of an individual trace logger work.

For example, suppose component A consists of 10 classes. Suppose each class is configured to use a separate trace logger. All 10 trace loggers in the component are registered as members of the same group, for example, Component_A_Group. You can turn on trace for a single class, or you can turn on trace for all 10 classes in a single operation using the group name, if you want a component trace. Group names are maintained within the namespace for trace loggers.

## JRas framework (deprecated)

Because the JRas extensions classes do not provide the flexibility and behavior that are required for many scenarios, a variety of extension points are defined. You can write your own implementation classes to obtain the required behavior.

**Deprecated:** The JRas framework described in this topic is deprecated. However, you can achieve similar results using Java logging.

In general, the JRas extensions require you to call the Manager class to obtain a message logger or trace logger. No provision is made for you to provide your own message or trace logger subclasses. In general, user-provided extensions cannot be used to affect the integrated mode of operation. The behavior of the integrated mode of operation is solely determined by the WebSphere Application Server run time and the JRas extensions classes.

### Handlers

The stand-alone JRas logging toolkit defines the RASIHandler interface. All handlers must implement this interface. You can write your own handler classes that implement the RASIHandler interface. Directly create instances of user-defined handlers and add them to the loggers that are obtained from the Manager class.

The stand-alone JRas logging toolkit provides several handler implementation classes. These handler classes are inappropriate for use in the Java 2 Platform, Enterprise Edition (J2EE) environment. You cannot directly use or subclass any of the Handler classes that are provided by the stand-alone JRas logging toolkit. Doing so is a violation of the programming model.

### Formatters

The stand-alone JRas logging toolkit defines the RASIFormatter interface. All formatters must implement this interface. You can write your own formatter classes that implement the RASIFormatter interface. You can add these classes to a user-defined handler only. WebSphere Application Server handlers cannot be configured to use user-defined formatters. Instead, directly create instances of your formatters and add them to the your handlers appropriately.

As with handlers, the stand-alone JRas logging toolkit provides several formatter implementation classes. Direct use of these formatter classes is not supported.

### Message event types

The stand-alone JRas toolkit defines message event types in the RASIMessageEvent interface. In addition, the WebSphere Application Server reserves a range of message event types for future use. The RASIMessageEvent interface defines three types, with values of `0x01`, `0x02`, and `0x04`. The values `0x08` through `0x8000` are reserved for future use. You can provide your own message event types by extending this interface appropriately. User-defined message types must have a value of `0x1000` or greater.

Message loggers that are retrieved from the Manager class have their message masks set to `pass` or process all message event types defined in the RASIMessageEvent interface. To process user-defined message types, you must manually set the message logger mask to the appropriate state by user code after the message logger is obtained from the Manager class. WebSphere Application Server does not provide any built-in systems management support for managing message types.

### Message event objects

The stand-alone JRas toolkit provides a RASMessageEvent implementation class. When a message logging method is called on the message logger, and the message type is currently enabled, the logger creates and distributes an event of this class to all handlers that are currently registered with that logger.

You can provide your own message event classes, but they must implement the RASIEvent interface. You must directly create instances of such user-defined message event classes. When it is created, pass your message event to the message logger by calling the message logger's fireRASEvent method directly. WebSphere Application Server message loggers cannot directly create instances of user-defined types in response to calling a logging method (`msg.message`) on the logger. In addition, instances of user-defined message types are never processed by the WebSphere Application Server handler. You cannot create instances of the RASMessageEvent class directly.

### Trace event types

The stand-alone JRas toolkit defines trace event types in the RASITraceEvent interface. You can provide your own trace event types by extending this interface appropriately. In such a case, you must ensure that the values for the user-defined trace event types do not collide with the values of the types that are defined in the RASITraceEvent interface.

Trace loggers that are retrieved from the Manager class typically have their trace masks set to reject all types. A different starting state can be specified by using WebSphere Application Server systems management facilities. In addition, you can change the state of the trace mask for a logger at run-time, using WebSphere Application Server systems management facilities.

To process user-defined trace types, the trace logger mask must be manually set to the appropriate state by user code. WebSphere Application Server systems management facilities cannot be used to manage user-defined trace types, either at start time or run time.

### Trace event objects

The stand-alone JRas toolkit provides a RASTraceEvent implementation class. When a trace logging method is called on the WebSphere Application Server trace logger and the type is currently enabled, the logger creates and distributes an event of this class to all the handlers that are currently registered with that logger.

You can provide your own trace event classes. Such trace event classes must implement the RASIEvent interface. You must create instances of such user-defined event classes directly. When it is created, pass the trace event to the trace logger by calling the trace logger's fireRASEvent method directly. WebSphere Application Server trace loggers cannot directly create instances of user-defined types in response to calling a trace method (`entry`, `exit`, `trace`) on the trace logger. In addition, instances of user-defined trace types are never processed by the WebSphere Application Server handler. You cannot create instances of the RASTraceEvent class directly.

### User defined types, user defined events and WebSphere Application Server

By definition, the WebSphere Application Server handler processed user-defined message or trace types, or user-defined message or trace event classes. Message and trace entries of either a user-defined type or user-defined event class cannot be written to the WebSphere Application Server run-time logs.

#### *JRas programming interfaces for logging (deprecated):*

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

**General considerations**

You can configure the WebSphere Application Server to use Java 2 security to restrict access to protected resources such as the file system and sockets. Because user-written extensions typically access such protected resources, user-written extensions must contain the appropriate security checking calls, using AccessController doPrivileged calls. In addition, the user-written extensions must contain the appropriate policy file. In general, locating user-written extensions in a separate package is a good practice. It is your responsibility to restrict access to the user-written extensions appropriately.

**Writing a handler**

User-written handlers must implement the RASIHandler interface. The RASIHandler interface extends the RASIMaskChangeGenerator interface, which extends the RASIObject interface. A short discussion of the methods that are introduced by each of these interfaces follows, along with implementation pointers. For more in-depth information on any of the particular interfaces or methods, see the corresponding product API documentation.

**RASIObject interface**

The RASIObject interface is the base interface for stand-alone JRas logging toolkit classes that are stateful or configurable, such as loggers, handlers, and formatters.
- The stand-alone JRas logging tookit supports rudimentary properties-file based configuration. To implement this configuration support, the configuration state is stored as a set of key-value pairs in a properties file. The public Hashtable getConfig and public void setConfig(Hashtable ht) methods are used to get and set the configuration state. The JRas extensions do not support properties-based configuration. Implement these methods as no-operations. You can implement your own properties-based configuration using these methods.
- Loggers, handlers, and formatters can be named objects. For example, the JRas extensions require the user to provide a name for the loggers that are retrieved from the manager. You can name your handlers. The public String getName and public void setName(String name) methods are provided to

get or set the name field. The JRas extensions currently do not call these methods on user handlers. You can implement these methods as you want, including as no operations.

- Loggers, handlers, and formatters can also contain a description field. The public String getDescription and public void setDescription(String desc) methods can be used to get or set the description field. The JRas extensions currently do not use the description field. You can implement these methods as you want, including as no operations.
- The public String getGroup method is provided for use by the RASManager interface. Since the JRas extensions provide their own Manager class, this method is never called. Implement this as a no-operation.

**RASIMaskChangeGenerator interface**

The RASIMaskChangeGenerator interface is the interface that defines the implementation methods for filtering of events based on a mask state. It is currently implemented by both loggers and handlers. By definition, an object that implements this interface contains both a message mask and a trace mask, although both need not be used. For example, message loggers contain a trace mask, but the trace mask is never used because the message logger never generates trace events. Handlers, however, can actively use both mask values. For example, a single handler can handle both message and trace events.

- The public long getMessageMask and public void setMessageMask(long mask) methods are used to get or set the value of the message mask. The public long getTraceMask and public void setTraceMask(long mask) methods are used to get or set the value of the trace mask.

In addition, this interface introduces the concept of *calling back* to interested parties when a mask changes state. The callback object must implement the RASIMaskChangeListener interface.

- The public void addMaskChangeListener(RASIMaskChangeListener listener) and public void removeMaskChangeListener(RASIMaskChangeListener listener) methods are used to add or remove listeners to the handler. The public Enumeration getMaskChangeListeners method returns an enumeration over the list of currently registered listeners. The public void fireMaskChangedEvent(RASMaskChangeEvent mc) method is used to call back all the registered listeners to inform them of a mask change event.

For efficiency reasons, the JRas extensions message and trace loggers implement the RASIMaskChangeListener interface. The logger implementations maintain a composite mask in addition to the logger mask. The logger composite mask is formed by logically *or'ing* the appropriate masks of all handlers that are registered to that logger, then *and'ing* the result with the logger mask. For example, the message logger composite mask is formed by or'ing the message masks of all handlers that are registered with that logger, then and'ing the result with the logger message mask.

All handlers are required to properly implement these methods. In addition, when a user handler is instantiated, the logger that is added must be registered with the handler; use the addMaskChangeListener method. When either the message mask or trace mask of the handler is changed, the logger must be called back to inform it of the mask change. With this process, the logger can dynamically maintain the composite mask.

The RASMaskChangedEvent class is defined by the stand-alone JRas logging toolkit. Direct use of that class by user code is supported in this context.

In addition, the RASIMaskChangeGenerator interface introduces the concept of caching the names of all message and trace event classes that the implementing object process. The intent of these methods is to support a management program such as a graphical user interface to retrieve the list of names, introspect the classes to determine the event types that they might possibly process and display the results. The JRas extensions do not ever call these methods, so they can be implemented as no operations.

- The public void addMessageEventClass(String name) and public void removeMessageEventClass(String name) methodscan be called to add or remove a message event class name from the list. The method public Enumeration getMessageEventClasses returns an enumeration over the list of message event class names. Similarly, the public void

addTraceEventClass(String name) and public void removeTraceEventClass(String name) methods can be called to add or remove a trace event class name from the list. The public Enumeration getTraceEventClasses method returns an enumeration over the list of trace event class names.

**RASIHandler interface**

The RASIHandler interface introduces the methods that are specific to the behavior of a handler.

The RASIHandler interface, as provided by the stand-alone JRas logging toolkit, supports handlers that run in either a synchronous or asynchronous mode. In asynchronous mode, events are typically queued by the calling thread and then written by a worker thread. Because spawning of threads is not supported in the WebSphere Application Server environment, it is expected that handlers do not queue or batch events, although this activity is not expressly prohibited.

- The public int getMaximumQueueSize() and public void setMaximumQueueSize(int size) methods create `IllegalStateException` exceptions to manage the maximum queue size. The public int getQueueSize method is provided to query the actual queue size.
- The public int getRetryInterval and public void setRetryInterval(int interval) methods support the notion of error retry, which implies some type of queueing.
- The public void addFormatter(RASIFormatter formatter), public void removeFormatter(RASIFormatter formatter) and public Enumeration getFormatters methods are provided to manage the list of formatters that the handler can be configured with. Different formatters can be provided for different event classes, if appropriate.
- The public void openDevice, public void closeDevice and public void stop methods are provided to manage the underlying device that the handler abstracts.
- The public void logEvent(RASIEvent event) and public void writeEvent(RASIEvent event) methods are provided to pass events to the handler for processing.

**Writing a formatter**

User-written formatters must implement the RASIFormatter interface. The RASIFormatter interface extends the RASIObject interface. The implementation of the RASIObject interface is the same for both handlers and formatters. A short discussion of the methods that are introduced by the RASIFormatter interface follows. For more in-depth information on the methods introduced by this interface, see the corresponding product API documentation.

**RASIFormatter interface**

- The public void setDefault(boolean flag) and public boolean isDefault methods are used by the concrete RASHandler classes that are provided by the stand-alone JRas logging toolkit to determine if a particular formatter is the default formatter. Because these RASHandler classes must never be used in a WebSphere Application Server environment, the semantic significance of these methods can be determined by the user.
- The public void addEventClass(String name), public void removeEventClass(String name) and public Enumeration getEventClasses methods are provided to determine which event classes a formatter can use to format. You can provide the appropriate implementations.
- The public String format(RASIEvent event) method is called by handler objects and returns a formatted String representation of the event.

## Programming model summary

The programming model that is described in this section builds upon and summarizes some of the concepts already introduced. This section also formalizes usage requirements and restrictions. Use of the WebSphere Application Server JRas extensions in a manner that does not conform to the following programming guidelines is prohibited.

**Deprecated:** The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

You can use the WebSphere Application Server JRas extensions in three distinct operational modes. The programming models concepts and restrictions apply equally across all modes of operation.

- You must not use implementation classes that are provided by the stand-alone JRas logging toolkit directly, unless specifically noted otherwise. Direct usage of those classes is not supported. IBM Support provides no diagnostic aid or bug fixes relating to the direct use of classes that are provided by the stand-alone JRas logging toolkit.
- You must obtain message and trace loggers directly from the Manager class. You cannot directly instantiate loggers.
- You cannot replace the WebSphere Application Server message and trace logger classes.
- You must guarantee that the logger names that are passed to the Manager class are unique, and follow the documented naming constraints. When a logger is obtained from the Manager class, you must not attempt to change the name of the logger by calling the setName method.
- Named loggers can be used more than once. For any given name, the first call to the Manager class results in the Manager class creating a logger that is associated with that name. Subsequent calls to the Manager class that specify the same name result in a returned reference to the existing logger.
- The Manager class maintains a hierarchical namespace for loggers. Use a dot-separated, fully qualified class name to identify any logger. Other than dots or periods, logger names cannot contain any punctuation characters, such as an asterisk (*), a comma (.), an equals sign (=), a colon (:), or quotes.
- Group names must comply with the same naming restrictions as logger names.
- The loggers returned from the Manager class are subclasses of the RASMessageLogger and the RASTraceLogger classes that are provided by the stand-alone JRas logging toolkit. You can call any public method that is defined by the RASMessageLogger and RASTraceLogger classes. You cannot call any public method that is introduced by the provided subclasses.
- If you want to operate in either stand-alone or combined mode, you must provide your own Handler and Formatter subclasses. You cannot use the Handler and Formatter classes that are provided by the stand-alone JRas logging toolkit. User written handlers and formatters must conform to the documented guidelines.
- Loggers that are obtained from the Manager class come with a WebSphere Application Server handler installed. This handler writes message and trace records to logs that are defined by the WebSphere Application Server run time. Manage these logs using the provided systems management interfaces.
- You can programmatically add and remove user-defined handlers from a logger at any time. Multiple additions and removals of user defined handlers are supported. You are responsible for creating an instance of the handler to add, configuring the handler by setting the handler mask value and formatter appropriately, then adding the handler to the logger using the addHandler method. You are responsible for programmatically updating the masks of user-defined handlers, as appropriate.
- You might get a reference to the handler that is installed within a logger by calling the getHandlers method on the logger and processing the results. You must not call any methods on the handler that are obtained in this way. You can remove the WebSphere Application Server handler from the logger by calling the logger removeHandler method, passing in the reference to the WebSphere Application Server handler. When removed, the WebSphere Application Server handler cannot be added again to the logger.
- You can define your own message type. The behavior of user-defined message types and restrictions on their definitions is discussed in Extending the JRas framework.
- You can define your own message event classes. The use of user-defined message event classes is discussed in Extending the JRas framework.
- You can define your own trace types. The behavior of user-defined trace types and restrictions on your definitions is discussed in Extending the JRas framework.
- You can define your own trace event classes. The use of user-defined trace event classes is discussed in Extending the JRas framework.
- You must programmatically maintain the bits in the message and trace logger masks that correspond to any user-defined types. If WebSphere Application Server facilities are used to manage the predefined types, these updates must not modify the state of any of the bits that correspond to those types. If you are assuming ownership responsibility for the predefined types, then you can change all bits of the masks.

# JRas messages and trace event types

The basic JRas message and event types are not the same as those natively recognized by WebSphere Application Server, so the JRas types are mapped onto the types that are native to the runtime environment. You can control the way JRas message and trace events are processed using custom filters and message controls.

## Event types

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

The base message and trace event types that are defined by the stand-alone JRas logging toolkit are not the same as the native types that are recognized by the WebSphere Application Server run-time. Instead, the basic JRas types are mapped onto the native types. This mapping can vary by platform or edition. The mapping is discussed in the following section.

## Platform message event types

The message event types that are recognized and processed by the WebSphere Application Server runtime are defined in the RASIMessageEvent interface that is provided by the stand-alone JRas logging toolkit. These message types are mapped onto the native message types, as follows.

| WebSphere Application Server native type | JRas RASIMessageEvent type |
|---|---|
| Audit | TYPE_INFO, TYPE_INFORMATION |
| Warning | TYPE_WARN, TYPE_WARNING |
| Error | TYPE_ERR, TYPE_ERROR |

## Platform trace event types

The trace event types that are recognized and processed by the WebSphere Application Server run time are defined in the RASITraceEvent interface that is provided by the stand-alone JRas logging toolkit. The RASITraceEvent interface provides a rich and complex set of types. This interface defines both a simple set of levels, as well as a set of enumerated types.

- For a user who prefers a simple set of levels, the RASITraceEvent interface provides `TYPE_LEVEL1`, `TYPE_LEVEL2`, and `TYPE_LEVEL3`. The implementations provide support for this set of levels. The levels are hierarchical, enabling level 2 also enables level 1, enabling level 3 also enables levels 1 and 2.
- For users who prefer a more complex set of values that can be *OR'd* together, the RASITraceEvent interface provides `TYPE_API`, `TYPE_CALLBACK`, `TYPE_ENTRY_EXIT`, `TYPE_ERROR_EXC`, `TYPE_MISC_DATA`, `TYPE_OBJ_CREATE`, `TYPE_OBJ_DELETE`, `TYPE_PRIVATE`, `TYPE_PUBLIC`, `TYPE_STATIC`, and `TYPE_SVC`.

The trace event types are mapped onto the native trace types as follows:

Mapping WebSphere Application Server trace types to the JRas RASITraceEvent level types.

| WebSphere Application Server native type | JRas RASITraceEvent level type |
|---|---|
| Event | TYPE_LEVEL1 |
| EntryExit | TYPE_LEVEL2 |
| Debug | TYPE_LEVEL3 |

Mapping WebSphere Application Server trace types to the JRas RASITraceEvent enumerated types.

| WebSphere Application Server native type | JRas RASITraceEvent enumerated types |
|---|---|

| Event | TYPE_ERROR_EXC, TYPE_SVC, TYPE_OBJ_CREATE, TYPE_OBJ_DELETE |
| --- | --- |
| EntryExit | TYPE_ENTRY_EXIT, TYPE_API, TYPE_CALLBACK, TYPE_PRIVATE, TYPE_PUBLIC, TYPE_STATIC |
| Debug | TYPE_MISC_DATA |

For simplicity, it is recommended that one or the other of the tracing type methodologies is used consistently throughout the application. If you decide to use the non-level types, choose one type from each category and use those types consistently throughout the application, to avoid confusion.

## Message and trace parameters

The various message logging and trace method signatures accept the `Object`, `Object[]` and `Throwable` parameter types. WebSphere Application Server processes and formats the various parameter types as follows:

**Primitives**
: Primitives, such as int and long are not recognized as subclasses of Object type and cannot be directly passed to one of these methods. A primitive value must be transformed to a proper Object type (Integer, Long) before passing as a parameter.

**Object**
: The toString method is called on the object and the resulting String is displayed. Implement the toString method appropriately for any object that is passed to a message logging or trace method. It is the responsibility of the caller to guarantee that the toString method does not display confidential data such as passwords in clear text, and does not cause infinite recursion.

**Object[]**
: The Object[] type is provided for the case when more than one parameter is passed to a message logging or trace method. The toString method is called on each Object in the array. Nested arrays are not handled, that is none of the elements in the Object array belong in an array.

**Throwable**
: The stack trace of the Throwable type is retrieved and displayed.

**Array of primitives**
: An array of primitive, for example, `byte[]`, `int[]`, is recognized as an Object, but is treated somewhat as a second cousin of Object by Java code. In general, avoid arrays of primitives, if possible. If arrays of primitives are passed, the results are indeterminate and can change, depending on the type of array passed, the API used to pass the array, and the release of the product. For consistent results, user code needs to preprocess and format the primitive array into some type of String form before passing it to the method. If such preprocessing is not performed, the following problems can result:

- [B@924586a0b - This message is deciphered as a byte array at location X. This message is typically returned when an array is passed as a member of an Object[] type and results from calling the toString method on the byte[] type.
- Illegal trace argument : array of long. This response is typically returned when an array of primitives is passed to a method taking an Object.
- 01040703: The hex representation of an array of bytes. Typically this problem can occur when a byte array is passed to a method taking a single Object. This behavior is subject to change and cannot be relied on.
- ″1″ ″2″: The String representation of the members of an int[] type formed by converting each element to an integer and calling the toString method on the integers. This behavior is subject to change and cannot be relied on.
- [Ljava.lang.Object;@9136fa0b : An array of objects. Typically this response is seen when an array containing nested arrays is passed.

## Controlling message logging

Writing a message to a WebSphere Application Server log requires that the message type passes three levels of filtering or screening:
1. The message event type must be one of the message event types that is defined in the RASIMessageEvent interface.
2. Logging of that message event type must be enabled by the state of the message logger mask.
3. The message event type must pass any filtering criteria that is established by the WebSphere Application Server run-time.

When a WebSphere Application Server logger is obtained from the Manager class, the initial setting of the mask forwards all native message event types to the WebSphere Application Server handler. It is possible to control what messages get logged by programmatically setting the state of the message logger mask.

Some editions of the product support user specified message filter levels for a server process. When such a filter level is set, only messages at the specified severity levels are written to WebSphere Application Server. Message types that pass the mask check of the message logger can be filtered out by WebSphere Application Server.

## Control tracing

Each edition of the product provides a mechanism for enabling or disabling trace. The various editions can support static trace enablement (trace settings are specified before the server is started), dynamic trace enablement (trace settings for a running server process can be dynamically modified), or both.

Writing a trace record to a WebSphere Application Server requires that the trace type passes three levels of filtering or screening:
1. The trace event type must be one of the trace event types that is defined in the RASITraceEvent interface.
2. Logging of that trace event type must be enabled by the state of the trace logger mask.
3. The trace event type must pass any filtering criteria that is established by the WebSphere Application Server run-time.

When a logger is obtained from the Manager class, the initial setting of the mask is to suppress all trace types. The exception to this rule is the case where the WebSphere Application Server run time supports static trace enablement and a non-default startup trace state for that trace logger is specified. Unlike message loggers, the WebSphere Application Server can dynamically modify the trace mask state of a trace logger. WebSphere Application Server only modifies the portion of the trace logger mask that corresponds to the values that are defined in the RASITraceEvent interface. WebSphere Application Server does not modify undefined bits of the mask that might be in use for user-defined types.

When the dynamic trace enablement feature that is available on some platforms is used, the trace state change is reflected both in the application server run time and the trace mask of the trace logger. If user code programmatically changes the bits in the trace mask corresponding to the values that are defined by in the RASITraceEvent interface, the mask state of the trace logger and the run time state become unsynchronized and unexpected results occur. Therefore, programmatically changing the bits of the mask corresponding to the values that are defined in the RASITraceEvent interface is not supported.

**Related tasks**

"Programming with the JRas framework" on page 323
Use the JRas extensions to incorporate message logging and diagnostic trace into WebSphere Application Server applications.

# Instrumenting an application with JRas extensions

You can create an application using JRas extensions.

### Before you begin

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

To create an application using the WebSphere Application Server JRas extensions, perform the following steps:

1. Determine the mode for the extensions: integrated, stand-alone, or combined.
2. If the extensions are used in either stand-alone or combined mode, create the necessary handler and formatter classes.
3. If localized messages are used by the application, create a resource bundle.
4. In the application code, get a reference to the Manager class and create the manager and logger instances.
5. Insert the appropriate message and trace logging statements in the application.

## Creating JRas resource bundles and message files

The WebSphere Application Server message logger provides the message and msg methods so the user can log localized messages. In addition, the message logger provides the textMessage method to log messages that are not localized. Applications can use either or both, as appropriate.

#### Before you begin

The JRas framework that is described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

#### About this task

The mechanism for providing localized messages is the resource bundle support that is provided by the IBM Developer Kit, Java Technology Edition. If you are not familiar with resource bundles as implemented by the Developer Kit, you can get more information from various texts, or by reading the API documentation for the java.util.ResourceBundle, java.util.ListResourceBundle and java.util.PropertyResourceBundle classes, as well as the java.text.MessageFormat class.

The PropertyResourceBundle class is the preferred mechanism to use. In addition, note that the JRas extensions do not support the extended formatting options such as {1, date} or {0, number, integer} that are provided by the MessageFormat class.

You can forward messages that are written to the internal WebSphere Application Server logs to other processes for display. For example, messages that are displayed on the administrative console, which can be running in a different location than the server process, can be localized using the *late binding* process. Late binding means that WebSphere Application Server does not localize messages when they are logged, but defers localization to the process that displays the message.

To properly localize the message, the displaying process must have access to the resource bundle where the message text is stored. You must package the resource bundle separately from the application, and install it in a location where the viewing process can access it. If you do not want to take these steps, you can use the early binding technique to localize messages as they are logged.

The two techniques are described as follows:

**Early binding**

The application must localize the message before logging it. The application looks up the localized

text in the resource bundle and formats the message. When formatting is complete, the application logs the message using the textMessage method. Use this technique to package the application resource bundles with the application.

**Late binding**

The application can choose to have the WebSphere Application Server run time localize the message in the process where it displays. Using this technique, the resource bundles are packaged in a stand-alone `.jar` file, separately from the application. You must then install the resource bundle `.jar` file on every machine in the installation from which an administrative console or log viewing program might be run. You must install the `.jar` file in a directory that is part of the extensions class path. In addition, if you forward logs to IBM service, you must also forward the `.jar` file that contains the resource bundles.

To create a resource bundle, perform the following steps.

1. Create a text properties file that lists message keys and the corresponding messages. The properties file must have the following characteristics:
   - Each property in the file is terminated with a line-termination character.
   - If a line contains only white space, or if the first non-white space character of the line is the number sign symbol (#) or exclamation mark (!), the line is ignored. The # and ! characters can therefore be used to put comments into the file.
   - Each line in the file, unless it is a comment or consists only of white space, denotes a single property. A backslash (\) is treated as the line-continuation character.
   - The syntax for a property file consists of a key, a separator, and an element. Valid separators include the equal sign (=), colon (:), and white space ( ).
   - The key consists of all characters on the line from the first non-white space character to the first separator. Separator characters can be included in the key by escaping them with a backslash (\), but using this approach is not recommended because escaping characters is error prone and confusing. Instead, use a valid separator character that does not display in any keys in the properties file.
   - White space after the key and separator is ignored until the first non-white space character is encountered. All characters that remain before the line-termination character define the element.

   See the Java documentation for the java.util.Properties class for a full description of the syntax and construction of properties files.
2. Translate the file into localized versions of the file with language-specific file names for example, the DefaultMessages.properties file can be translated into `DefaultMessages_de.properties` for German and `DefaultMessages_ja.properties` for Japanese.
3. When the translated resource bundles are available, write them to a system-managed persistent storage medium. Resource bundles are used to convert the messages into the requested national language and locale.
4. When a message logger is obtained from the JRas manager, configure the logger to use a particular resource bundle. Messages logged through the `message` API use this resource bundle when message localization is performed. At run time, the user's locale setting is used to determine the properties file from which to extract the message that is specified by a message key, ensuring that the message is delivered in the correct language.
5. If the message loggers msg method is called, explicitly identify a resource bundle name.

## What to do next

The application locates the resource bundle based on the file location relative to any directory in the class path. For instance, if the DefaultMessages.properties property resource bundle is in the `baseDir/subDir1/subDir2/resources` directory and `baseDir` is in the class path, the name *subdir1.subdir2*.resources.DefaultMessage is passed to the message logger to identify the resource bundle.

***JRas resource bundles:***

You can create resource bundles in several ways. The best and easiest way is to create a properties file that supports a PropertiesResourceBundle resource bundle. This sample shows how to create such a properties file.

**Resource bundle sample**

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

For this sample, four localizable messages are provided. The properties file is created and the key-value pairs are inserted into it. All the normal properties files conventions and rules apply to this file. In addition, the creator must be aware of other restrictions that are imposed on the values by the Java MessageFormat class. For example, apostrophes must be escaped or they cause a problem. Avoid the use of non-portable characters. WebSphere Application Server does not support the use of extended formatting conventions that the MessageFormat class supports, such as {1, date} or {0, number, integer}.

Assume that the base directory for the application that uses this resource bundle is `baseDir` and that this directory is in the class path. Assume that the properties file is stored in the subdirectory `baseDir` that is not in the class path (*baseDir/subDir1/subDir2/resources*). To allow the messages file to resolve, the subDir1.subDir2.resources.DefaultMessage name is used to identify the PropertyResourceBundle resource bundle and is passed to the message logger.

For this sample, the properties file is named `DefaultMessages.properties`:

```
# Contents of the DefaultMessages.properties file
MSG_KEY_00=A message with no substitution parameters.
MSG_KEY_01=A message with one substitution parameter: parm1={0}
MSG_KEY_02=A message with two substitution parameters: parm1={0}, parm2 = {1}
MSG_KEY_03=A message with three substitution parameters: parm1={0}, parm2 = {1}, parm3={2}
```

When the `DefaultMessages.properties` file is created, the file can be sent to a translation center where the localized versions are generated.

## JRas manager and logger instances

You can use the JRas extensions in integrated, stand-alone, or combined mode. Configuration of the application varies depending on the mode of operation, but use of the loggers to log message or trace entries is identical in all modes of operation.

**Deprecated:** The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

Integrated mode is the default mode of operation. In this mode, message and trace events are sent to the WebSphere Application Server logs.

In the combined mode, message and trace events are logged to both WebSphere Application Server and user-defined logs.

In the stand-alone mode, message and trace events are logged only to user-defined logs.

### Using the message and trace loggers

Regardless of the mode of operation, the use of message and trace loggers is the same.

### Using a message logger

The message logger is configured to use the DefaultMessages resource bundle. Message keys must be passed to the message loggers if the loggers are using the message API.

```
msgLogger.message(RASIMessageEvent.TYPE_WARNING, this,
     methodName, "MSG_KEY_00");
... msgLogger.message(RASIMessageEvent.TYPE_WARN, this,
     methodName, "MSG_KEY_01", "some string");
```

If message loggers use the msg API, you can specify a new resource bundle name.

```
msgLogger.msg(RASIMessageEvent.TYPE_ERR, this, methodName,
                  "ALT_MSG_KEY_00", "alternateMessageFile");
```

You can also log a text message. If you are using the textMessage API, no message formatting is done.

```
msgLogger.textMessage(RASIMessageEvent.TYPE_INFO, this, methodName,"String and Integer",
"A String", new Integer(5));
```

### Using a trace logger

Because trace is normally disabled, guard trace methods for performance reasons.

```
private void methodX(int x, String y, Foo z)
{
   // trace an entry point. Use the guard to make sure tracing is enabled.
Do this checking before you gather parameters to  trace.
   if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT) {
        // I want to trace three parameters, package them up in an Object[]
        Object[] parms = {new Integer(x), y, z};
        trcLogger.entry(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX", parms);
  }
... logic
  // a debug or verbose trace point
  if (trcLogger.isLoggable(RASITraceEvent.TYPE_MISC_DATA) {
        trcLogger.trace(RASITraceEvent.TYPE_MISC_DATA, this, "methodX" "reached here");
  }
  ...
  // Another classification of trace event. An important state change is
 detected, so a different trace type is used.
  if (trcLogger.isLoggable(RASITraceEvent.TYPE_SVC) {
     trcLogger.trace(RASITraceEvent.TYPE_SVC, this, "methodX", "an important event");
  }
 ...
  // ready to exit method, trace. No return value to trace
    if (trcLogger.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
        trcLogger.exit(RASITraceEvent.TYPE_ENTRY_EXIT, this, "methodX");
   }
}
```

## Setting up for integrated JRas operation

Use JRas operations in integrated mode to send trace events and logging messages to only WebSphere Application Server logs.

### Before you begin

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

In the integrated mode of operation, message and trace events are sent to WebSphere Application Server logs. This approach is the default mode of operation.

1. Import the requisite JRas extensions classes:

   ```
   import com.ibm.ras.*;
   import com.ibm.websphere.ras.*;
   ```

2. Declare logger references:

```
private RASMessageLogger msgLogger = null;
private RASTraceLogger trcLogger = null;
```

3. Obtain a reference to the Manager class and create the loggers. Because loggers are named singletons, you can do this activity in a variety of places. One logical candidate for enterprise beans is the ejbCreate method. For example, for the myTestBean enterprise bean, place the following code in the ejbCreate method:

```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
        myTestBean.class.getName());



// Configure the message logger to use the message file that is created
// for this application.
msgLogger.setMessageFile("acme.widgets.DefaultMessages");
trcLogger = mgr.createRASTraceLogger("Acme", "Widgets", "RasTest",
        myTestBean.class.getName());
mgr.addLoggerToGroup(trcLogger, groupName);
```

## Setting up for combined JRas operation

Use JRas operation in combined mode to output trace data and logging messages to both WebSphere Application Server and user-defined logs.

### Before you begin

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

In combined mode, messages and trace are logged to both WebSphere Application Server logs and user-defined logs. The following sample assumes that:

- You wrote a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter.
- You are not using user-defined types or events.

1. Import the requisite JRas extensions classes:

```
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;
```

2. Import the user handler and formatter:

```
import com.ibm.ws.ras.test.user.*;
```

3. Declare the logger references:

```
private RASMessageLogger msgLogger = null;
  private RASTraceLogger trcLogger = null;
```

4. Obtain a reference to the Manager class, create the loggers, and add the user handlers. Because loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for enterprise beans is the ejbCreate method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must support this approach. The following sample is a message logger sample. The procedure for a trace logger is similar.

```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
    msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
           myTestBean.class.getName());
    // Configure the message logger to use the message file defined
    // in the ResourceBundle sample.
    msgLogger.setMessageFile("acme.widgets.DefaultMessages");

   // Create the user handler and formatter. Configure the formatter,
```

```
// then add it to the handler.
RASIHandler handler = new SimpleFileHandler("myHandler", "FileName");
RASIFormatter formatter = new SimpleFormatter("simple formatter");
 formatter.addEventClass("com.ibm.ras.RASMessageEvent");
 handler.addFormatter(formatter);

// Add the Handler to the logger. Add the logger to the list of the
//handlers listeners, then set the handlers
// mask, which updates the loggers composite mask appropriately.
// WARNING - there is an order dependency here that must be followed.
msgLogger.addHandler(handler);
handler.addMaskChangeListener(msgLogger);
handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

## Setting up for stand-alone JRas operation

You can configure JRas operations to output trace data and logging messages to only user-defined locations.

### Before you begin

The JRas framework described in this task and its sub-tasks is deprecated. However, you can achieve similar results using Java logging.

### About this task

In stand-alone mode, messages and traces are logged only to user-defined logs. The following sample assumes that:

- You have a user-defined handler named SimpleFileHandler and a user-defined formatter named SimpleFormatter.
- You are not using user-defined types of events.

1. Import the requisite JRas extensions classes:
```
import com.ibm.ras.*;
import com.ibm.websphere.ras.*;
```
2. Import the user handler and formatter:
```
import com.ibm.ws.ras.test.user.*;
```
3. Declare the logger references:
```
private RASMessageLogger msgLogger = null;
  private RASTraceLogger trcLogger = null;
```
4. Obtain a reference to the Manager class, create the loggers, and add the user handlers. Because loggers are named singletons, you can obtain a reference to the loggers in a number of places. One logical candidate for enterprise beans is the ejbCreate method. Make sure that multiple instances of the same user handler are not accidentally inserted into the same logger. Your initialization code must support this approach. The following sample is a message logger sample. The procedure for a trace logger is similar.
```
com.ibm.websphere.ras.Manager mgr = com.ibm.websphere.ras.Manager.getManager();
    msgLogger = mgr.createRASMessageLogger("Acme", "WidgetCounter", "RasTest",
             myTestBean.class.getName());
    // Configure the message logger to use the message file that is defined in
    //the ResourceBundle sample.
    msgLogger.setMessageFile("acme.widgets.DefaultMessages");

    // Get a reference to the Handler and remove it from the logger.
    RASIHandler aHandler = null;
    Enumeration enum = msgLogger.getHandlers();
    while (enum.hasMoreElements()) {
         aHandler = (RASIHandler)enum.nextElement();
         if (aHandler instanceof WsHandler)
              msgLogger.removeHandler(wsHandler);
    }
```

```
 // Create the user handler and formatter. Configure the formatter,
 // then add it to the handler.
 RASIHandler handler = new SimpleFileHandler("myHandler", "FileName");
 RASIFormatter formatter = new SimpleFormatter("simple formatter");
 formatter.addEventClass("com.ibm.ras.RASMessageEvent");
 handler.addFormatter(formatter);

// Add the Handler to the logger. Add the logger to the list of the
// handlers listeners, then set the handlers
// mask, which will update the loggers composite mask appropriately.
// WARNING - there is an order dependency here that must be followed.
msgLogger.addHandler(handler);
handler.addMaskChangeListener(msgLogger);
handler.setMessageMask(RASIMessageEvent.DEFAULT_MESSAGE_MASK);
```

## Logging Common Base Events in WebSphere Application Server

WebSphere Application Server uses Common Base Events within its logging framework. Common Base Events can be created explicitly and then logged through the Java logging API, or can be created implicitly by using the Java logging API directly.

### About this task

An *event* is a notification from an application or the application server that reports information that is related to a specific problem or situation. Common Base Events provide you with a standard structure for these event notifications, which allow you to correlate events that are received from different applications. Log Common Base Events to capture events from different sources to help you fix a problem within an application environment or to tune system performance.

For Common Base Event creation, the application server environment provides a Common Base Event factory with a content handler that provides both runtime data and template data for Common Base Events.

1. Optional: Read about the Common Base Event types and how they are implemented within an application server. Refer to "The Common Base Event in WebSphere Application Server."
2. Read "Logging Common Base Events in WebSphere Application Server" on page 367.
3. Configure the Common Base Event framework for your application server using one of the following methods:
   - "Logging with Common Base Event API and the Java logging API" on page 356
   - "Generate Common Base Event content with the default event factory" on page 357.

### Results

Common Base Events will now be logged according to your configuration. Use these event logs to determine the source of application problems.

## The Common Base Event in WebSphere Application Server

The Common Base Event is an XML document that defines a common representation of events that is intended for use by enterprise management and business applications. The Common Base Event defines common fields, the values they can take, and the exact meanings of these values.

An application creates an event object whenever something happens that either needs to be recorded for later analysis or which might require the trigger of additional work. An *event* is a structured notification that reports information that is related to a situation. An event reports three kinds of information:
- The situation: What happened
- The identity of the affected component: For example, the server that shut down

- The identity of the component that is reporting the situation, which might be the same as the affected component

The application that creates the event object is called the *event source*. Event sources can use a common structure for the event. The accepted standard for such a structure is called the *Common Base Event*. The Common Base Event is an XML document that is defined as part of the autonomic computing initiative.

The Common Base Event model is a standard that defines a common representation of events that is intended for use by enterprise management and business applications. This standard, which is developed by the IBM Autonomic Computing Architecture Board, supports encoding of logging, tracing, management, and business events using a common XML-based format. This format makes it possible to correlate different types of events that originate from different applications. For more information about the Common Base Event model, see the Common Base Event specification (*Canonical Situation Data Format: The Common Base Event V1.0.1*). The common event infrastructure currently supports Version 1.0.1 of the specification.

The basic concept behind the Common Base Event model is the *situation*. A situation can be anything that happens anywhere in the computing infrastructure, such as a server shutdown, a disk-drive failure, or a failed user login. The Common Base Event model defines a set of standard situation types that accommodate most of the situations that might arise (for example, StartSituation and CreateSituation).

The Common Base Event contains all of the information that is needed by the consumers to understand the event. This information includes data about the runtime environment, the business environment, and the instance of the application object that created the event.

For complete details on the Common Base Event format, see the XML schema that is included in the Common Base Event specification document, at http://www.ibm.com/developerworks/autonomic/books/fpy0mst.htm#HDRCBEDESC .

## Types of problem determination events

Problem determination involves multiple types of data, including at least two different classes of event data, log events, and diagnostic events.

Log events, which are also referred to as *message events*, are typically emitted by components of a business application during normal deployment and operations. Log events might identify problems, but these events are also normally available and emitted while an application and its components are in production mode. The target audience for log and message events is users and administrators of the application and the components that make up the application. Log events are normally the only events available when a problem is first detected, and are typically used during both problem recovery and problem resolution.

Diagnostic events, which are commonly referred to as *trace events*, are used to capture internal diagnostic information about a component, and are usually not emitted or available during normal deployment and operation. The target audience for diagnostic events is the developers of the components that make up the business application. Diagnostic events are typically used when trying to resolve problems within a component, such as a software failure, but are sometimes used to diagnose other problems, especially when the information provided by the log events is not sufficient to resolve the problem. Diagnostic events are typically used when trying to resolve a problem.

A *Common Base Event* is a common structure for an event. It defines common fields, the values that these fields can take, and the exact meanings of these values for an event. Common Base Events are primarily used to represent log events.

## Common Base Event structure

A *Common Base Event* is a common structure for an event. It defines common fields, the values that these fields can take, and the exact meanings of these values for an event.

The Common Base Event contains several structural elements. These elements include:

- Common header information
- Component identification, both source and reporter
- Situation information
- Message data
- Extended data
- Context data
- Associated events and association engine

Each of these structural elements has its own embedded elements and attributes.

The following table presents a summary of all the fields in the Common Base Event and their usage requirements for problem determination events. This table shows whether a particular element or attribute is required, recommended, optional, prohibited, or discouraged for log events, and the base specification.

| Field name | Log events | Base specification |
|---|---|---|
| Version | Required | Required |
| creationTime | Required | Required |
| severity | Required | Optional |
| Msg | Required | Optional |
| sourceComponentId* | Required | Required |
| sourceComponentId.location | Required | Required |
| sourceComponentId.locationType | Required | Required |
| sourceComponentId.component | Required | Required |
| sourceComponentId.subComponent | Required | Required |
| sourceComponentId.componentIdType | Required | Required |
| sourceComponentId.componentType | Required | Required |
| sourceComponentId.application | Recommended | Optional |
| sourceComponentId.instanceId | Recommended | Optional |
| sourceComponentId.processId | Recommended | Optional |
| sourceComponentId.threadId | Recommended | Optional |
| sourceComponentId.executionEnvironment | Optional | Optional |
| situation* | Required | Required |
| situation.categoryName | Required | Required |
| situation.situationType* | Required | Required |
| situation.situationType.reasoningScope | Required | Required |
| situation.situationType.(specific Situation Type elements) | Required | Required |
| msgDataElement* | Recommended | Optional |
| msgDataElement .msgId | Recommended | Optional |
| msgDataElement .msgIdType | Recommended | Optional |
| msgDataElement .msgCatalogId | Recommended | Optional |
| msgDataElement .msgCatalogTokens | Recommended | Optional |
| msgDataElement .msgCatalog | Recommended | Optional |
| msgDataElement .msgCatalogType | Recommended | Optional |
| msgDataElement .msgLocale | Recommended | Optional |

| extensionName | Recommended | Optional |
|---|---|---|
| localInstanceId | Optional | Optional |
| globalInstanceId | Optional | Optional |
| priority | Discouraged | Optional |
| repeatCount | Optional | Optional |
| elapsedTime | Optional | Optional |
| sequenceNumber | Optional | Optional |
| reporterComponentId* | Optional | Optional |
| reporterComponentId.location | Required (2) | Required (2) |
| reporterComponentId.locationType | Required (2) | Required (2) |
| reporterComponentId.component | Required (2) | Required (2) |
| reporterComponentId.subComponent | Required (2) | Required (2) |
| reporterComponentId.componentIdType | Required (2) | Required (2) |
| reporterComponentId.componentType | Required (2) | Required (2) |
| reporterComponentId.instanceId | Optional | Optional |
| reporterComponentId.processId | Optional | Optional |
| reporterComponentId.threadId | Optional | Optional |
| reporterComponentId.application | Optional | Optional |
| reporterComponentId.executionEnvironment | Optional | Optional |
| extendedDataElements* | Note 3 | Optional |
| contextDataElements* | Note 4 | Optional |
| associatedEvents* | Note 5 | Optional |

**Notes:**

- Items followed by an asterisk (*) are elements that consist of sub elements and attributes. The fields in those elements are listed in the table directly following the parent element name.
- Some of the elements are optional, but when included, they include sub elements and attributes that are required. For example, the reporterComponentId element has a ComponentIdentification type. The component attribute in ComponentIdentification is required. Therefore, the reporterComponentId.component attribute is required, but only when the reporterComponentId parent element is included.
- The extendedDataElements element can be included multiple times to supply extended data information. See the Extended data section for more information on required and recommended extended data element values.
- The contextDataElements element can be included multiple times to supply context data information.
- The associatedEvents element can be included multiple times to supply correlation data. No recommended uses of this element exist for the producers of problem determination data, and the use of this element is discouraged.

***Common header information:***

This topic provides additional information about how to format and use these fields for problem determination events, which can be used to clarify and extend the information provided in the other documents.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

The common header information in the Common Base Event includes the following information about an event:

- Version: The version of this Common Base Event
- creationTime: The date and time when the event generated
- Severity and priority: The severity of the condition (situation) that is identified by the event
- extensionName: The type of event that was captured
- localInstanceId and globalInstanceId: Identifiers that can be used to quickly identify a specific event within a set of events
- repeatCount and elapsedTime: Information that supports a system to efficiently report multiple events of the same type, by consolidating those events into a single event
- sequenceNumber: Sequence information that supports a system to order a set of events in other ways than time of capture

**severity**

All problem determination events must provide an indication as to the relative severity of the condition (situation) being reported by providing appropriate values for the severity field in the Common Base Event. The severity field is required for problem determination events. This field is more restrictive than the base specification for the Common Base Event, which lists this field as optional because effective and efficient problem determination requires the ability to quickly identify the information that is needed to resolve a problem as well as prioritize the problems that need addressing. Typically, the following values are used for problem determination events:

| | | |
|---|---|---|
| 10 | Information | Log information events, normal conditions, and events that are supplied to clarify operations, for example, state transitions, operational changes. These events typically do not require administrator action or intervention. |
| 20 | Harmless | Similar to information events, but are used to capture audit items, such as state transitions or operational changes. These events typically do not require administrator action or intervention. |
| 30 | Warning | Warnings typically represent recoverable errors, for example a failure that the system can correct. These events can require administrator action or intervention. |
| 40 | Minor | Minor errors describe events that represent an unrecoverable error within a component. The failure affects the component ability to service some requests. The business application can continue to perform its normal functions, but its overall operation might be degraded. These events require administrator action or intervention to address the condition. |

| | | |
|---|---|---|
| 50 | Critical | Critical errors describe events that represent an unrecoverable error within a component. The failure significantly affects the component ability to service most requests. The business application can continue most, but not all of its normal functions and its overall operation might be degraded. These events require administrator action or intervention to address the condition. |
| 60 | Fatal | Fatal errors describe events that represent an unrecoverable error within a component. The failure usually results in the complete failure of the component. The business application can continue some normal functions, but its overall operation might be degraded. These events require administrator action or intervention to address the condition. |

**msg**
Refer to “Message data” on page 352 for information on this attribute.

**priority**
The use of the priority field is discouraged for problem determination events. The severity field is typically used to communicate and evaluate the importance of problem determination events. Use the priority field to enhance the information that is provided in the severity field, that is. prioritize events of the same severity.

**extensionName**
The extensionName field is used to communicate the type of event that is reported, for example, what general class of events is being reported. In many cases this field provides an indication of what additional data you can expect with the event, for example, optional data values.

**repeatCount**
The repeatCount field is valid for problem determination events, but is not typically used or supplied by the event producers. This field is used for data reduction and consolidation by event management and analysis systems.

**elapsedTime**
The elapsedTime field is valid for problem determination events, but is not typically used or supplied by the event producers. This field is used for data reduction and consolidation by event management and analysis systems.

**sequenceNumber**
The sequenceNumber field is valid for problem determination events. It is typically used only by event producers when the granularity of the event time stamp (the creationTime field) is not sufficient in ordering events. The sequenceNumber field is typically used to sequence events that have the same time stamp value.

Event management and analysis systems can use the sequenceNumber field for a number of reasons, including providing alternative sequencing, not necessarily based on a time stamp. The recommendations here are provided primarily for event producers.

***Component identification for source and reporter:***

The component identification fields in the Common Base Event are used to indicate which component in the system is experiencing the condition that is described by the event (the sourceComponentID) and which component emitted the event (the reporterComponentID).

Typically, these components are the same, in which case only the sourceComponentID is supplied. Some notes and scenarios on when to use these two elements in the Common Base Event:

- The sourceComponentID is always used to identify the component experiencing the condition that is described by the event.
- The reporterComponentID is used to identify the component that actually produced and emitted the event. This element is typically used only within events that are emitted by a component that is monitoring another component and providing operational information regarding that component. The monitoring component (for example, a Tivoli® agent or hardware device driver) is identified by the reporterComponentID and the component being monitored (for example, a monitored server or hardware device) is identified by the sourceComponentID.

  A potential misuse of the reporterComponentID is to identify a component that provides event conversion or management services for a component, for example, identifying an adapter that transforms the events that are captured by a component into Common Base Event format. The event conversion function is considered an extension of the component and not identified separately.

The information that is used to identify a component in the system is the same, regardless of whether it is the source component or reporter component:

| | | |
|---|---|---|
| location locationType | Component location | Identifies the location of the component. |
| component componentIdType | Component name | Identifies the asset name of the component, as well as the type of component. |
| subcomponent | Subcomponent name | Identifies a specific part or subcomponent of a component, for example a software module or hardware part. |
| application | Business application name | Identifies the business application or process the component is a part of and provides services for. |
| instanceId | Operational instance | Identifies the operational instance of a component, that is the actual running instance of the component. |
| processId threadId | Operational instance | Identifies the operational instance of a component within the context of a software operating system, that is he operating system process and thread running when the event was produced. |
| executionEnvironment | Operational instance Component location | Provides additional information about the operational instance of a component or its location by identifying the name of the environment hosting the operational instance of the component, for example the operating system name for a software application, the application server name for a Java 2 Platform, Enterprise Edition (J2EE) application, or the hardware server type for a hardware part. |

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines. This section provides additional information about how to format and use some of these fields for problem determination events, which can be used to clarify and extend the information that is provided in the other documents.

**Component**
The Component field in a problem determination event is used to identify the manageable asset that is associated with the event. A manageable asset is open for interpretation, but a good working definition is a manageable asset represents a hardware or software component that can be separately obtained or developed, deployed, managed, and serviced. Examples of typical component names are:

- IBM eServer™ xSeries® model x330
- IBM WebSphere Application Server version 5.1 (5.1 is the version number)
- Microsoft Windows 2000
- The name of an internally developed software application for a component

**subComponent**
The Subcomponent field in a problem determination event identifies the specific part of a component that is associated with the event. The subcomponent name is typically not a manageable asset, but provides internal diagnostic information when diagnosing an internal defect within a component, that is What part failed? Examples of typical subcomponents and their names are:

- Intel Pentium® processor within a server system (Intel Pentium IV Processor)
- the enterprise bean container within a Web application server (enterprise bean container)
- the task manager within an operating system (Linux Kernel Task Manager)
- the name of a Java class and method (myclass.mycompany.com or myclass.mycompany.com.methodname).

The format of a subcomponent name is determined by the component, but use the convention shown previously for naming a Java class or the combination of a Java class and method is followed. The subcomponent field is required in the Common Base Event.

**componentIdType**
The componentIdType field is required by the Common Base Event specification, but provides minimal value for problem determination events. For most problem determination events, it is encouraged to use the value provided in the application field instead of the componentIdType. The componentIdType field identifies the type of component; the application is identified by the application field.

**application**
The application field is listed as an optional value within the Common Base Event specification, but provide it within problem determination events whenever it this value is available. The only reason this field is not required for problem determination events is that instances exist where the issuing component might not be aware of the overall business application.

**instanceId**
The instanceId field is listed as an optional value within the Common Base Event specification, but provide this value within problem determination events whenever it is available.

Always provide the instanceID when a software component is identified and identify the operational instance of the component (for example, which operation instance of an installed software image is actually associated with the event). Provide this value for hardware components when these components support the concept of operational instances.

The format of the supplied value is defined by the component, but must be a value that an analysis system can use (either human or programmatic) to identify the specific running instance of the identified component. Examples include:

- **cell, node, server** name for the IBM WebSphere Application Server
- **deployed EAR file name** for a Java enterprise bean

- **serial number** for a hardware processor

**processId**
: The processId field is listed as an optional value within the Common Base Event specification, but provide this value for problem determination events whenever it is available and applicable. Always provide this value for software-generated events, and identify the operating system process that is associated with the component that is identified in the event. Match the format of the thread ID with the format of the operating system (or other running environment, such as a Java virtual machine). This field is typically not applicable or used for events that are emitted by hardware (for example, firmware).

**threadId**
: The threadId field is listed as an optional value within the Common Base Event specification, but provide this value for problem determination events whenever it is available and applicable. Always provide for software-generated events, and identify the active operating system thread when the event was detected or issued. A notable exception to this recommendation is some operating systems or running environments do not support threads. Match the format of the thread ID with the format of the operating system (or other running environment, such as a Java virtual machine). This field is typically not applicable or used for events that are emitted by hardware (for example, firmware).

**executionEnvironment**

The executionEnvironment field, when used, identifies the immediate running environment that is used by the component being identified. Some examples are:

- the operating system name when the component is a native software application.
- the operating system/Java virtual machine name when the component is a Java 2 Platform, Standard Edition (J2SE) application.
- the Web server name when the component is a servlet.
- the portal server name when the component is a portlet.
- the application server name when the component is an enterprise bean.

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

***Situation information:***

The situation information is used to classify the condition that is reported by an event into a common set of situations.

The Common Base Event specification [CBE101] provides information on the set of situations defined for the Common Base Event, with the values and formats that are used to describe these situations. The Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines.

Consider the following points regarding situation information for problem determination events:

- Whenever possible, use the situation categorizations and qualifiers that are described in the base Common Base Event specification. Avoid using your own situation definitions as much as possible.
- Not all messages and logs can be classified using the situation definitions that are supplied in the base Common Base Event specification. You can use the OtherSituation categorization to provide your own situation information, but the recommended course of action for problem determination events is to use the ReportSituation categorization, with reportCategory=Log.
- Warning events can be confusing. A warning event (that is an event with severity=warning) typically indicates a recoverable failure, but the situation settings can be interpreted as unrecoverable failures (for example ConnectSituation, successDisposition=UNSUCCESSFUL). Use the appropriate situation categorization so the severity setting indicates the severity of the situation, that is whether the component recovered from the failure.
- The recommended setting for the reasoningScope value is EXTERNAL for all message events.

***Message data:***

All problem determination Common Base Events must provide human readable text that describes the specific reported event within the msg field of the Common Base Event.

The text that is associated with events representing actual messages or log entries is expected to be translated and localized. Include the msgDataElement element in the Common Base Event whenever internationalized text is provided in the event. This element provides information about how the message text is created and how to interpret it. This information is particularly invaluable when trying to interpret the event programmatically or when trying to interpret the message independent of the locale or language that is used to format the message text.

**Prerequisite:** Understand the concepts that are associated with creating internationalized messages. A good source of education on these concepts is provided by the documentation that is associated with internationalization of Java information and the usage of resource bundles within the Java language.

The msgDataElement element in the Common Base Event includes the following information about the value of the msg field that is provided with an event:
- The locale of the supplied message text, which identifies how the locale-independent fields within the message are formatted, as well as the language of the message (msgLocale).
- A locale-independent identifier that is associated with the message that can be used to interpret the message independent of the message language, message locale, and message format (msgId and msgIdType).
- Information on how a translated message is created, including:
  - The identifier that is used to retrieve the message template (msgCatalogId).
  - The name and type of message catalog that are used to retrieve the message template (msgCatalog and msgCatalogType).
  - Any locale-independent information that is inserted into the message template to create the final message (msgCatalogTokens).

The Common Base Event specification [CBE101] provides information on the required format of these fields and the Common Base Event Developer's Guide [CBEBASE] provides general usage guidelines. This section provides additional information about how to format and use these fields for problem determination events.

**msg**

All message, log, and trace events must provide a human-readable message in the msg field of the Common Base Event. The msg field is required for problem determination events, both log events and diagnostic events. This field is more restrictive than the base specification for the Common Base Event, which lists this field as optional; effective and efficient problem determination requires the ability to quickly identify the reported condition. The format and usage of this message is component-specific, but use the following general guidelines:

- Expect the message text that is supplied with messages and log events to be internationalized.
- Provide the locale of the supplied message text with the msgLocale field in the msgDataElement element of the Common Base Event.
- Provide additional information regarding the format and construction of internationalized messages whenever possible, using the msgDataElement element of the Common Base Event.

**msgLocale**

Provide the message locale whenever message text is provided within the Common Base Event, as is the case with all problem determination events. The msgLocale field is listed as an optional value within the Common Base Event specification, but provide this information within problem determination events whenever possible. The reason this field is not required for problem determination events is that instances exist where the locale information is not provided or available when formatting the Common Base Event.

**msgId and msgIdType**

Several companies include a locale-independent identifier within internationalized message text that you can use to interpret the described condition by the message text, independent of the message. For example, most messages issued by IBM software look like `IEE890I WTO Buffers in console backup storage = 1024`, where a unique, locale-independent identifier `IEE890I` precedes the translated message text. This identifier provides a way to uniquely detect and identify a message independent of location and language. This detection is invaluable for locale-independent and programmatic analysis.

The msgId field is listed as an optional value within the Common Base Event specification, but it must be provided within problem determination events whenever this identifier is included in the message text. Likewise, the msgIdType field is listed as an optional value within the Common Base Event specification, but it must be provided within problem determination events whenever a value is supplied for msgId. Do not supply these fields when the message text is not translated or localized, for example, for trace events.

**msgCatalogId**

The msgCatalogId field is listed as an optional value within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. This field is not required for problem determination events because not all problem determination events include translated message text Some cases exist where the value is not provided or available when formatting the Common Base Event. Do not supply this field when the message text is not translated or localized, for example, for trace events.

**msgCatalogTokens**

The msgCatalogTokens field is listed as an optional value within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. This field is not required for problem determination events because not all problem determination events include translated message text, and cases exist where the value is not provided or available when formatting the Common Base Event. This value contains the list of locale-independent values or message tokens that are inserted into the localized message text when creating a translated message.

These values are difficult to extract from a translated message without knowing the translated message template that is used to create the message. Do not supply this field when the message text is not translated or localized

The Common Base Event provides several mechanisms for providing additional data about an event, including this field, extended data elements, and extensions to the schema. Always use the msgCatalogTokens field to supply the list of message tokens that is included in the message text associated with an event. These values can also be supplied in other parts of the Common Base Event, but they must be included in this field.

**msgCatalog and msgCatalogType**

The msgCatalog and msgCatalogType fields are listed as optional values within the Common Base Event specification, but provide this value whenever the Common Base Event includes localized or translated message text, for example when providing problem determination events that represent issued messages or log events. These fields are not required for problem determination events because not all problem determination events include translated message text, and cases exist where the values are not provided or available when formatting the Common Base Event. Do not complete these fields when the message text has is not translated or localized, for example, for trace events.

***Extended data:***

The Common Base Event provides several methods for including this additional data, including extending the Common Base Event schema or supplying one or more ExtendedDataElement elements within the Common Base Event, which is the preferred approach.

The base information that is included in a Common Base Event might not be sufficient to represent all of the information captured by a component when creating a problem determination event.

Use an ExtendedDataElement element to represent a single data item. A Common Base Event can contain more than one of these elements, essentially one for each additional data item. A hint to the number and type of ExtendedDataElement elements is supplied by the extensionName value, but this information is only a hint. The usage of the attributes in the ExtendedDataElement element for problem determination events is the same as those for any other Common Base Event.

## Sample Common Base Event instance

This XML document is an example of a Common Base Event instance that is generated by a WebSphere Application Server application.

Use the following example for reference:

```
<CommonBaseEvent  creationTime="2004-09-18T04:03:28.484Z"
   globalInstanceId="myhost:1095479647062:1899"
   msg="WSVR0024I: Server server1 stopped"
   severity="10"
   version="1.0.1">

 ... several extendedDataElements for WebSphere Application Server internal use only ...

<sourceComponentId  component="com.ibm.ws.runtime.component.ServerCollaborator"
    componentIdType="Unknown"
    executionEnvironment="Windows 2000[x86]#5.0"
    instanceId="myhost\myhost\server1"
    location="myhost"
    locationType="Hostname"
    processId="1095479647062"
    subComponent="Unknown"
    threadId="Alarm : 0"
    componentType="http://www.ibm.com/namespaces/autonomic/WebSphereApplicationServer"/>

 <msgDataElement msgLocale="en_US">
  <msgCatalogTokens value="server1"/>
  <msgId>WSVR0024I< /msgId>
  <msgCatalogId>WSVR0024I< /msgCatalogId>
  <msgCatalog>com.ibm.ws.runtime.runtime< /msgCatalog>
 </msgDataElement>

 <situation categoryName="ReportSituation">
  <situationType xsi:type="ReportSituation" reasoningScope="EXTERNAL" reportCategory="LOG"/>
 </situation>

</CommonBaseEvent>
```

A number of extendedDataElement elements in the XML are used by WebSphere Application Server, but are not for application use because these elements might change.

The CommonBaseEvent element defines the Common Base Event instance. This element has a set of attributes that are common for all Common Base Events. This set includes the extensionName attribute, which defines the type or class of the Common Base Event instance, the creation time, severity, and priority.

Nested within the CommonBaseEvent element are elements giving more detail about the situation. The first of these elements is the situation element. This classification is standardized.

The CommonBaseEvent element also includes the sourceComponentId and the (optional) reporterComponentId elements. The sourceComponentId element describes where the situation occurred; the reporterComponentId describes where the situation is detected. If the sourceComponentId and the reporterComponentId elements are the same, the reporterComponentId element is omitted.

The attributes of both the sourceComponentId and the reporterComponentId elements are the same. They identify the component type, name, operating system, and network location. The content of these attributes provides vertical correlation of the stack of IT resources that are active when the Common Base Event is created.

Also included in the CommonBaseEvent element are contextDataElements elements that describe the context in which the situation occurred. This context correlates Common Base Event instances that are part of the same work. This correlation is called *horizontal correlation* because an instance of a particular context type correlates events at the same level of abstraction, for example at the business level, the application level, or at the middleware level.

FExtended data elements contain additional data that is used to describe a situation. In this example, an extended data element is added by WebSphere Application Server to describe the Java 2 Platform, Enterprise Edition (J2EE) component that generated the Common Base Event instance and some application data.

## Sample Common Base Event template

The content handler uses template information to fill in blanks in the Common Base Event when the Common Base Event complete method is called.

Components that use the WebSphere Application Server event factory home can include a Common Base Event template XML file to provide data to populate Common Base Events. Information that is already supplied in the event is not overridden if the same field is supplied in the template.

The following example illustrates a Common Base Event template:

```
<?xml version="1.0" encoding="UTF-8"?>

<TemplateEvent
  version="1.0.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

 <CommonBaseEvent
  <sourceComponentId application="My Application" component="com.ibm.componentX"/>
  <extendedDataElements name="Sample ExtendedDataElement name" type="string">
   <values>Sample ExtendedDataElement value</values>
  </extendedDataElements>
 </CommonBaseEvent>

</TemplateEvent>
```

## Component identification for problem determination

This topic describes types of problem determination events.

A business application is made up of multiple components. A component can be made up of several internal subcomponents. Consistent application of these concepts is critical for effective problem determination of a business application; all of the parts of the application must use the same concepts and assumptions when creating and formatting events. Use the following definitions and examples when creating Common Base Events for problem determination.

**Business application**

A business application is the business logic and business data that is used to address a set of specific business requirements. A business application consists of several components of multiple types, combined in a unique manner by an enterprise, to provide the functions and resources that are needed to address those requirements. The primary creator and manager of a business application is the enterprise, and each enterprise or company creates unique business applications. Examples of business applications are the Payroll Application for the ACME Corporation and the Inventory Application for Spacely Sprockets.

**Components**

A business application is created and managed by the enterprise as a set of components. Components are deployable assets, which are developed either by the enterprise or a vendor, and managed by the enterprise. A component might be created by the enterprise, typically for use within a specific business application. For example, the ACME Corporation might create a set of enterprise beans to represent the business logic that is required by their Payroll Application. A component might also be an asset that is produced by a vendor and acquired by an enterprise. Examples of these components are hardware products, such as IBM eServers or Sun Solaris systems, or software products, such as IBM WebSphere Application Server, Oracle Database Servers.

**Subcomponents**

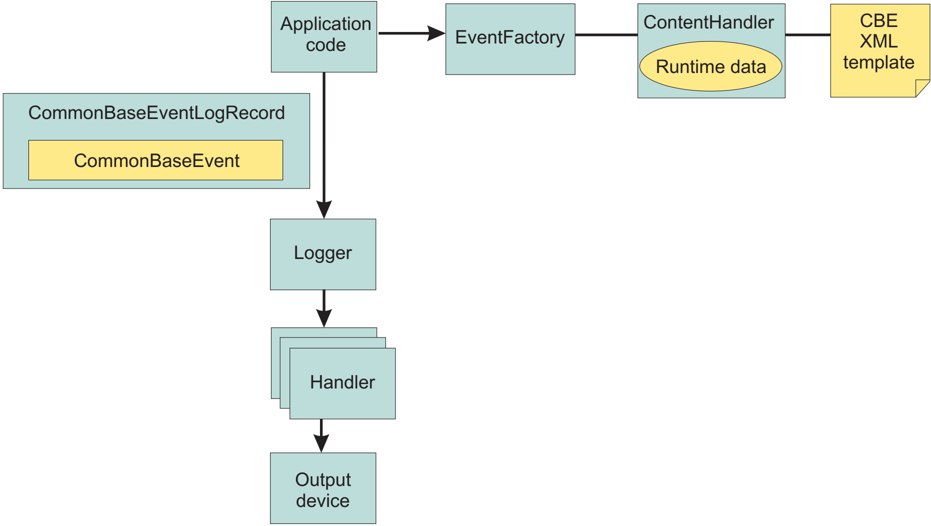
A specific component, depending on its complexity, might consist of several subcomponents. For example, the IBM WebSphere Application Server consists of many subcomponents, such as the enterprise bean container and the servlet engine. Subcomponent information is typically used only by the creator of the component to service the component, and as such are not separately deployable or manageable resources in the enterprise. The enterprise might deploy a change or update to a subcomponent, but only upon guidance from the component vendor and as part of the vendor's component. For example, a software fix for the enterprise bean container of the IBM WebSphere Application Server is packaged and deployed as a software update to the IBM WebSphere Application Server. Replacement of the processor in an IBM eServer is deployed as a physical part, but only as a part of the original deployed component, the IBM eServer.

## Logging with Common Base Event API and the Java logging API

In cases where the events that are generated by the Java logging API are insufficient to describe the event that needs capturing, you can create Common Base Events with the Common Base Event factory APIs.

### Before you begin

When you create a Common Base Event, you can add data to the Common Base Event before it is logged. The following diagram illustrates how application code can create and log Common Base Events:




### About this task

WebSphere Application Server is configured to use an event factory that automatically populates WebSphere Application Server-specific information into the Common Base Events that it generates. In general, it is good practice to create events using the WebSphere Application Server default Common

Base Event factory because this approach ensures consistency of Common Base Event content across events. However, you can create and use other Common Base Event factories.

Common Base Events are initiated and logged in the following sequence:

1. Application code invokes the createCommonBaseEvent method on the EventFactory class to create a CommonBaseEvent.
2. Application code wraps CommonBaseEvent event in a CommonBaseEventLogRecord record, and adds event-specific data.
3. Application code calls the CommonBaseEvent event complete method.
4. The CommonBaseEvent event invokes the ContentHandler completeEvent method.
5. The ContentHandler handler adds XML template data to the CommonBaseEvent event. Not all ContentHandler handlers support templates.
6. The ContentHandler handler adds runtime data to the CommonBaseEvent event.
7. Application code passes the CommonBaseEventLogRecord record to the logger using the Logger.log method.
8. Logger passes CommonBaseEventLogRecord record to Handlers.
9. Handlers format data and write to the output device.

- You can use the default Common Base Event factory to generate content. Read "Generate Common Base Event content with the default event factory" for more information.
- If you do not wish to use the default event factory, you can create custom content handlers and event factories.
  1. Create a custom factory home. Read "Creating custom Common Base Event factory homes" on page 362.
  2. Create a custom content handler. Read "Creating custom Common Base Event content handlers" on page 360.

## Results

After completing all the above steps you will have a Common Base event based on your configuration settings.

## Generate Common Base Event content with the default event factory

A default Common Base Event content handler populates Common Base Events with WebSphere Application Server runtime information. This content handler can also use a Common Base Event template to populate Common Base Events.

The default content handler is used when the server creates CommonBaseEventLogRecords as would be the case in the following example:

```
// Get a named logger
Logger logger = Logger.getLogger("com.ibm.someLogger");
// Log to the logger -- implicitly the default content handler
// will be associated with the CommonBaseEvent contained in the
// CommonBaseEventLogRecord.
logger.warning("MSG_KEY_001");
```

To specify a Common Base Event template in the above case, a Logger.properties file would need to be provided with an eventfactory entry for *com.ibm.someLogger*. If a valid template is found on the classpath, then the Logger's event factory will use the specified template's content in addition to the WebSphere Application Server runtime information when populating Common Base Events. If the template is not found on the classpath, or is invalid, then the Logger's event factory will only use the WebSphere Application Server runtime information when populating Common Base Events.

The default content handler is also associated with the event factory home supplied in the global event factory context. This is convenient for creating Common Base Events that need to be populated with content similar to that generated from the WebSphere Application Server:

```
// Request the event factory from the global event factory home
EventFactory eventFactory =
   EventFactoryContext.getInstance().getEventFactoryHome().getEventFactory(templateName);

// Create a Common Base Event
CommonBaseEvent commonBaseEvent = eventFactory.createCommonBaseEvent();

// Complete the Common Base Event using content from the template (if specified above)
// and the server runtime information.
eventFactory.getContentHandler().completeEvent(commonBaseEvent);
```

In the above example, if the template referenced by *templateName* is found on the classpath, and the template is valid, then the event factory home will return an event factory which uses a content handler that combines the template's content with the WebSphere Application Server runtime information when populating Common Base Events. If the template is not found on the classpath, or is invalid, then the event factory home will return an event factory which uses a content handler that uses only the WebSphere Application Server runtime information when populating Common Base Events.

The default content handler populates Common Base Events in the server environment with the following runtime information:

**CommonBaseEvent.globalInstanceId**
: **Value:** The *unique_record_id*

  Set this value only if the CommonBaseEvent.globalInstanceId value is null before the completeEvent method is called.

**CommonBaseEvent.msg**
: **Value:** A localized message that is based on the MsgDataElement element.

  Set this value only if the CommonBaseEvent.msg message is null before the completeEvent method is called.

**CommonBaseEvent.severity**
: **Value:** Set based on the value of level set on the CommonBaseEventLogRecord record, if level >= Level.SEVERE, set to `50`; if level >= Level.WARNING, set to `30`; the default is set to `10`.

  Set this value only if the CommonBaseEvent.severity value is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.component**
: **Value:**Set based on the LoggerName value that is set on the CommonBaseEventLogRecord record.

  Set this value only if the CommonBaseEvent.ComponentIdentification.component is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.componentIdType**
: **Value:** "Unknown"

  Set this value only if the CommonBaseEvent.ComponentIdentification.componentIdType value is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.executionEnvironment**
: **Value:** `OSname[OSarch]#OSversion`

  Set this value only if the CommonBaseEvent.ComponentIdentification.executionEnvironment value is null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.instanceId**
: **Value:** `cellName\nodeName\serverName`

Set this value only if the CommonBaseEvent.ComponentIdentification.instanceId value is null before the completeEvent method is called. Set only in a server environment because this value is ignored in a client application.

**CommonBaseEvent.ComponentIdentification.location**
**Value:**The host name

Set this value only if both the CommonBaseEvent.ComponentIdentification.location and the CommonBaseEvent.ComponentIdentification.locationType values are null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.locationType**
**Value:** The host name

Set this value only if both the CommonBaseEvent.ComponentIdentification.location and the CommonBaseEvent.ComponentIdentification.locationType values are null before the completeEvent method is called.

**CommonBaseEvent.ComponentIdentification.processId**
**Value:** An internally generated representation of the process number.

Set this value only if the CommonBaseEvent.ComponentIdentification.processId value is null before the completeEvent method is called

**CommonBaseEvent.ComponentIdentification.subComponent**
**Value:** Set based on values of the sourceClassName and the sourceMethodName names that are set on the sourceClassName.sourceMethodName name of the CommonBaseEventLogRecord record.

Set this value only if the CommonBaseEvent.ComponentIdentification.subComponent values is null before the completeEvent method is called and both the sourceClassName and the sourceMethodName names are set.

**CommonBaseEvent.ComponentIdentification.threadId**
**Value:** Set to the value of the Java Virtual Machine (JVM) thread name.

Set this value only if the CommonBaseEvent.ComponentIdentification.threadId values is null before the completeEvent value is called.

**CommonBaseEvent.ComponentIdentification.componentType**
**Value:** http://www.ibm.com/namespaces/autonomic/WebSphereApplicationServer

Set this value only if the CommonBaseEvent.ComponentIdentification.componentType values is null before the completeEvent method is called.

**CommonBaseEvent.MsgDataElement.msgLocale**
**Value:** Set based on the default locale of the JVM.

Set this value only if the CommonBaseEvent.msg value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.categoryName**
**Value:** `ReportSituation`

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.situationType.type**
**Value:** `ReportSituation`

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.situationType.reasoningScope**
**Value:** EXTERNAL

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

**CommonBaseEvent.Situation.situationType.reportCategory**
**Value:** LOG

Set this value only if the CommonBaseEvent.Situation value is null before the completeEvent method is called.

The sourceComponentIdentification value is populated if no reporterComponentIdentification ID exists when the completeEvent method is invoked on the content handler. Otherwise, the reporterComponentIdentification ID is populated instead.

## Common Base Event content handler

Content handlers populate data into Common Base Events when the Common Base Event complete method is invoked. You can associate content handlers with Common Base Event templates, which provide default information to transfer into each Common Base Event.

Content handlers might also provide any other information that is relevant to completing the population of the Common Base Event, such as appropriate runtime defaults. The use of content handlers ensures consistency of field use in the Common Base Event within a component or within a set of components that share the same runtime. For example, some content handlers support the specification of a template. If used consistently across a component, this template ensures that all events for that component have the same template information filled in. Similarly, some content handlers can also supply runtime information to their associated Common Base Events. If consistently used throughout the entire runtime, runtime information ensures that all events use runtime data in a similar way.

The event factory home that is used in the WebSphere Application Server runtime is associated with a content handler that both reads from a template, and supplies runtime data. Have components use Event Factories that are obtained from this event factory home with their own templates, to produce consistency between application events and server events.

More details can be found in“Creating custom Common Base Event content handlers” or the API documentation for org.eclipse.hyades.logging.events.cbe.ContentHandler at www.eclipse.org/hyades.

## Creating custom Common Base Event content handlers

Create a custom Common Base Event content handler or template to automate configuration or values for specific events.

### Before you begin

A *content handler* is an object that automatically sets the property values of each event based on any arbitrary policies that you want to use.

The following content handler classes were added to WebSphere Application Server to facilitate the use of the Common Base Event infrastructure:

| Class Name | Description |
|---|---|
| WsContentHandlerImpl | This provides an implementation of org.eclipse.hyades.logging.events.cbe.ContentHandler specifically for use in the WebSphere Application Server environment. This content handler completes Common Base Events using information from the WebSphere Application Server runtime, and it uses the same content handler as is used internally by the WebSphere Application Server when completing Common Base Events for logging. |

| WsTemplateContentHandlerImpl | This provides the same function as WsContentHandlerImpl, but it extends the org.eclipse.hyades.logging.events.cbe.impl.TemplateContentHandlerImpl class to enable the use of a Common Base Event template. Template content takes precedence in cases where the template data specifies values for the same Common Base Event fields as does the WsContentHandlerImpl. |
|---|---|

## About this task

In some situations, you might want some event property data set automatically for every event that you create. This automation is a way to fill in certain standard values that do not change, such as the application name, or to set some properties based on information that is available from the runtime environment, like creation time or thread information. You can set property data automatically by creating a content handler.

- Use the following code sample to implement the CustomContentHandler class:

```
public class CustomContentHandler extends WsContentHandlerImpl {

 public CustomContentHandler() {
  super();
  // TODO Custom initialization code goes here
 }

 public void completeEvent(CommonBaseEvent cbe) throws CompletionException {
  // following code will add WAS content to the Content Base Event
  super.completeEvent(cbe);
  // TODO Custom content can be added to the Content Base Event here
 }
}
```

- The following shows how to implement the CustomTemplateContentHandler class:

```
public class CustomTemplateContentHandler extends WsTemplateContentHandlerImpl {

 public CustomTemplateContentHandler() {
  super();
  // TODO Custom initialization code goes here
 }

 public void completeEvent(CommonBaseEvent cbe) throws CompletionException {
  // following code will add WAS content to the Content Base Event
  super.completeEvent(cbe);
  // TODO Custom content can be added to the Content Base Event here
 }
}
```

## Results

You now have a content handler or a custom content handler template based on the settings that you specified.

# Common Base Event factory home

Event Factory homes provide Event Factory instantiation that is based on a unique factory name.

Event factory home implementations are tightly coupled with content handlers that are used to populate Common Base Events with template or default data. Event factory instances are maintained by the associated event factory home, based on their unique name. For example, when application code requests a named event factory, the newly created Event Factory instance is returned and persisted for future requests for that named event factory. An abstract event factory home class provides the implementation

for the APIs in the event factory home interface. Implementers extend the abstract event factory home class and implement the createContentHandler API to create a typed content handler that is based on the type of event factory home implementation.

In WebSphere Application Server, the default event factory home that is obtained with a call to EventFactoryContext.getInstance.getEventFactoryHome method is associated with a ContentHandler handler capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactoryHome at www.eclipse.org/hyades.

## Creating custom Common Base Event factory homes

Use custom Common Base Event factory homes to control configuration and implementation of unique event factories.

### Before you begin

Event factory homes create and provide homes for Event Factory instances. Each event factory home has a content handler. This content handler is assigned to every event factory the event factory home creates. In turn, when a Common Base Event is created, the content handler from the event factory is assigned to it. Event factory instances are maintained by the associated event factory home, based on their unique name. For example, when application code requests a named event factory, the newly created event factory instance is returned and persisted for future requests for that named event factory.

The following classes were added to facilitate the use of event eactory homes for logging Common Base Events:

| Class Name | Description |
| --- | --- |
| WsEventFactoryHomeImpl | This class extends the org.eclipse.hyades.logging.events.cbe.impl.AbstractEventFactoryHome class. This event factory home returns event factory instances associated with the WsContentHandlerImpl content handler. The WsContentHandlerImpl is the content handler used by the WebSphere Application Server by default when no event factory template is in use. |
| WsTemplateEventFactory HomeImpl | This class extends the org.eclipse.hyades.logging.events.cbe.impl.EventXMLFileEventFactoryHomeImpl class. This event factory home returns event factory instances associated with the WsTemplateContentHandlerImpl Content Handler. The WsTemplateContentHandlerImpl is the content handler used by the WebSphere Application Server when an Event Factory template is required. |

### About this task

Custom event factory homes support the use of Common Base Event for logging in WebSphere Application Server and make logging easy and consistent between the WebSphere Application Server runtime and the exploiters of this API. The CustomEventFactoryHome and CustomTemplateEventFactoryHome classes will be used to obtain an event factory. These classes are there to make sure the correct content handler is being used with a particular event factory. The CustomEventFactoryHelper class is an example of how the infrastructure provider can hide the factory selection details from infrastructure users, using their own set of parameters to decide which the appropriate event factory is.

- The following code samples provide examples of how to implement and use the CustomEventFactoryHome class.
  1. Implementation of the CustomEventFactoryHome class is as follows:

```
public class CustomEventFactoryHome extends AbstractEventFactoryHome {

 public CustomEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomContentHandler();
 }
}
```

2. The following is an example of how to use the CustomEventFactoryHome class:

```
// get the event factory
  EventFactory eventFactory=(new CustomEventFactoryHome()).getEventFactory("XYZ");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

- For the CustomTemplateEventFactoryHome class you can use the following code for implementation and use:
  1. Implement the CustomTemplateEventFactoryHome class by using this code:

```
public class CustomTemplateEventFactoryHome extends
  EventXMLFileEventFactoryHomeImpl {

 public CustomTemplateEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomTemplateContentHandler();
 }
}
```

  2. Use the CustomTemplateEventFactoryHome class by following this sample code:

```
// get the event factory
  EventFactory eventFactory=(new
    CustomTemplateEventFactoryHome()).getEventFactory("XYZ");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

- The CustomEventFactoryHelper class can be implemented and used by following the code below:
  1. Implement the custom CustomEventFactoryHelper class using this code:

```
public class CustomTemplateEventFactoryHome extends
  EventXMLFileEventFactoryHomeImpl {

 public CustomTemplateEventFactoryHome() {
  super();
  // TODO Custom intialization code goes here
 }

 public ContentHandler createContentHandler(String arg0) {
```

```
  // Always use custom content handler
  return resolveContentHandler();
 }

 public ContentHandler resolveContentHandler() {
  // Always use custom content handler
  return new CustomTemplateContentHandler();
 }
}
```

Figure 4 CustomTemplateEventFactoryHome class

```
public class CustomEventFactoryHelper {
 // name of the event factory to use
 public static final String FACTORY_NAME="XYZ";

 public static EventFactory getEventFactory(String param1, String param2) {
  EventFactory factory=null;
  switch (resolveFactory(param1,param2)) {
  case 1:
   factory=(new CustomEventFactoryHome()).getEventFactory(FACTORY_NAME);
   break;
  case 2:
   factory=(new
    CustomTemplateEventFactoryHome()).getEventFactory(FACTORY_NAME);
   break;

  default:
   // Add default for event factory
   break;
  }
  return factory;
 }

 private static int resolveFactory(String param1, String param2) {
  int factory=0;
  // Add code here to resolve which factory to use
  return factory;
 }
}
```

2. To use the CustomEventFactoryHelper class, use the following code:

```
// get the event factory
  EventFactory eventFactory=
   CustomEventFactoryHelper.getEventFactory("param1","param2","param3");
  // create an event - call appropriate method
  eventFactory.createCommonBaseEvent();
  // log event ...
```

### Results

Use the information provided here to implement a custom content factory home and the associated classes based on the settings that you specify.

## Common Base Event factory context

The event factory context provides a service to look up event factory homes. Retrieve the event factory context using a call to the EventFactoryContext.getInstance method.

Using this class, you can look up the event factory homes by name, and avoid the need to include the typed home in code. The EventFactoryHome name must be located on the class path to be found. The EventFactoryContext context also stores an EventFactoryHome name as a default, which can be obtained with a call to the EventFactoryContext.getInstance.getEventFactoryHome method.

In WebSphere Application Server, the EventFactoryContext context is configured with a default EventFactoryHome name which is associated to a ContentHandler handler that is capable of supplying both event template information, as well as WebSphere Application Server runtime default information.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactory at www.eclipse.org/hyades.

### Common Base Event factory

Use event factories to create Common Base Events and complete event properties with associated content handlers.

Content handlers populate data into Common Base Events when the Common Base Event invokes the complete method. All event properties set by the application code have priority over all properties that are specified by the content handler. Event factory implementations are tightly coupled with the content handler instance, which is associated with the event factory when the event factory is instantiated. Factory instances can be retrieved only from their associated event factory home. Event factory instances are retrieved and maintained based on unique names. Event factory names are hierarchical; they are represented using the standard Java dot-delimited, name-space naming conventions.

More details can be found in the API documentation for org.eclipse.hyades.logging.events.cbe.EventFactory at www.eclipse.org/hyades.

## java.util.logging -- Java logging programming interface

The java.util.logging.Logger class provides a variety of methods with which data can be logged.

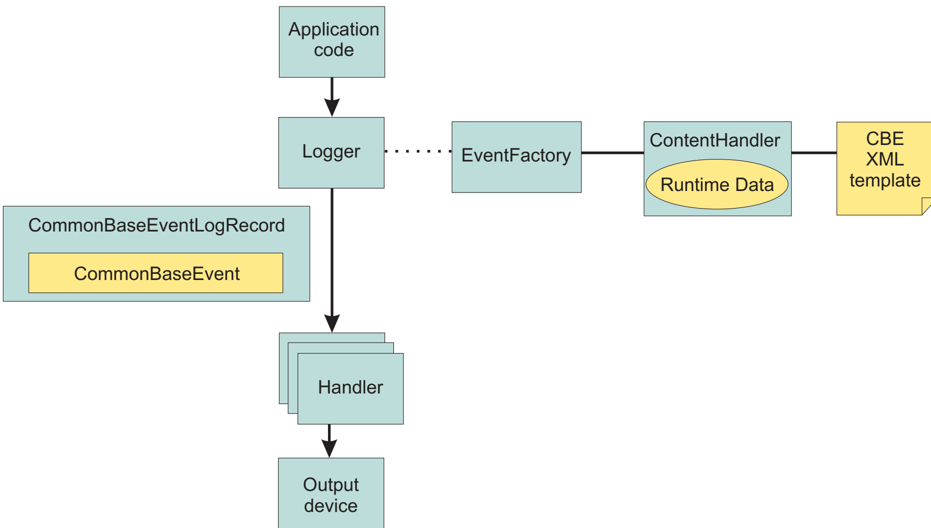
In the WebSphere Application Server, the Java logging API (java.util.logging) automatically creates Common Base Events for events that are logged at the WsLevel.DETAIL level or above (including WsLevel.DETAIL, Level.CONFIG, Level.INFO, WsLevel.AUDIT, Level.WARNING, Level.SEVERE, and WsLevel.FATAL). These Common Base Events are created using the event factory that is associated with the logger to which the message is logged. If no event factory is specified, WebSphere Application Server uses a default event factory which automatically fills in WebSphere Application Server-specific information.

The WebSphere Application Server uses a special implementation of the java.util.logging.Logger class that automatically creates Common Base Events for the following methods:

- config
- info
- warning
- severe
- log: All variants except log(LogRecord) when used with the WsLevel.DETAIL level or more severe levels
- logp: When used with the WsLevel.DETAIL level or more severe levels
- logrb: When used with the WsLevel.DETAIL level or more severe levels

The WebSphere Application Server logger implementation is used only for named loggers for example, loggers that are instantiated with calls, such as Logger.getLogger(″com.xyz.SomeLoggerName″). Loggers instantiated with calls to the Logger.getAnonymousLogger and Logger.getLogger, or Logger.global methods do not use the WebSphere Application Server implementation, and do not automatically create Common Base Events for logging requests made to them. Log records that are logged directly with the Logger.log(LogRecord) method are not automatically converted by WebSphere Application Server loggers into Common Base Events.

The following diagram illustrates how application code can log Common Base Events:




The Java logging API processing of named loggers and message-level events proceeds as follows:

1. Application code invokes the named logger (WsLevel.DETAIL or above) with event-specific data.
2. The logger creates a Common Base Event using the createCommonBaseEvent method on the event factory that is associated with the logger.
3. The logger creates a Common Base Event using the event factory associated to the logger.
4. The logger wraps the common base event in a CommonBaseEventLogRecord record, and adds event-specific data.
5. The logger calls the Common Base Event complete method.
6. The Common Base Event invokes the ContentHandler completeEvent method.
7. The content handler adds XML template data to the Common Base Event (including for example, the component name). Not all content handlers support templates.
8. The content handler adds runtime data to the Common Base Event (including for example, the current thread name).
9. The logger passes the CommonBaseEventLogRecord record to the handlers.
10. The handlers format data and write to the output device.

## Logger.properties file

Use the Logger.properties file to set logger attributes for your component.

The properties file is loaded the first time the Logger.getLogger(loggername) method is called within an application. The Logger.properties file must be either on the WebSphere Application Server class path, or the context class path.

The logging subsystem uses Common Base Events to represent all the messages in the WebSphere Application Server activity.log file. You can specify your own event factory template to be used with your loggers. Use the eventfactory property in your Logger.properties file. See “Sample Common Base Event template” on page 355 for details on the Common Base Event template.

By convention, the name of the event factory template file should be the fully qualified package name of the package using the template. The name of the file must end with the `.event.xml` extension. For example, a valid event factory template file name for the com.abc.somepackage package is:

```
com.abc.somepackage.event.xml
```

When you specify the property value for the eventfactory property in the Logger.properties file, include the full path name with no leading slash relative to the root of your class path entry. Do not include the `.event.xml` extension.

For example, if the template files from the example above are located in the `com/abc/templates` directory, the valid value for the eventfactory property is:

```
com/abc/templates/com.abc.somepackage
```

Finally, if this event factory template file is used by the com.abc.somepackage.SomeClass logger, then the following entry will appear in the Logger.properties file:

```
com.abc.somepackage.SomeClass.eventfactory=com/abc/templates/com.abc.somepackage
```

## Logging Common Base Events in WebSphere Application Server

The following practices ensure consistent use of Common Base Events within your components, and between your components and WebSphere Application Server components.

Follow these guidelines:

- Use a different logger for each component. Sharing loggers across components gets in the way of associating loggers with component-specific information.
- Associate loggers with event templates that specify source component identification. This association ensures that the source of all events created with the logger is properly identified.
- Use the same template for directly created Common Base Events (events created using the Common Base Event factories) and indirectly created Common Base Events (events created using the Java logging API) within the same component.
- Avoid calling the complete method on Common Base Events until you are finished adding data to the Common Base Event and are ready to log it. This approach ensures that any decisions made by the content handler based on data already in the event are made using the final data.

The following sample `Logger.properties` file entry demonstrates how to associate the com.ibm.componentX logger with the com.ibm.componentX event factory:

```
com.ibm.componentX.eventfactory=com.ibm.componentX
```

The following sample code demonstrates the use of the same event factory setting for direct (Part 1) and indirect (Part 2) Common Base Event logging:

```
<?xml version="1.0" encoding="UTF-8"?>

<TemplateEvent>
  version="1.0.1"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation="templateEvent.xsd">

 <CommonBaseEvent>
   <sourceComponentId application="My application" component="com.ibm.componentX"/>
   <extendedDataElements CommonBaseEventname="Sample ExtendedDataElement name" type="string">
   <values>Sample ExtendedDataElement value</values>
  </extendedDataElements>
 < /CommonBaseEvent>

< /TemplateEvent>
```

# Chapter 11. Securing Web services applications using the WSS APIs at the message level

Standards and profiles address how to provide protection for messages that are exchanged in a Web service environment. Web services security is a message-level standard that is based on securing SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens.

## Before you begin

To secure Web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, delegation, and auditing across a spectrum of application and business topologies. One of the key requirements for the security model in today's business environment is the ability to interoperate between formerly incompatible security technologies in heterogeneous environments. The complete Web services security protocol stack and technology roadmap is described in the Web services roadmap.

## About this task

The Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security: SOAP Message Security Version 1.1 specification is the basic messaging transport for all Web services. SOAP 1.2 adds extensions to the existing SOAP 1.1 extensions so that you can build secure Web services. Attachments can be added to SOAP messages by using Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP) instead of the SOAP with Attachments (SWA) profile.

The OASIS Web Services Security (WS-Security) Version 1.1 specification is the building block that is used in conjunction with other Web service and application-specific protocols to accommodate a wide variety of security models. Web services security for WebSphere Application Server is based on specific standards that are included in the OASIS Web Services Security Version 1.1 specification and profiles.

The Version 1.1 specification defines additional facilities for protecting the integrity and confidentiality of a message. The Version 1.1 specification also provides the mechanisms for associating security-related claims with the message. The Web Services Security Version 1.1 standards that are supported by WebSphere Application Server include the signature confirmation, encrypted header elements, the Username Token Profile and the X.509 Token Profile. The Username Token Profile and the X.509 Token Profile have been updated as Version 1.1 profiles. For the X.509 Certificate Token Profile, one new type of security token reference is the Thumbprint reference, which is specified in the binding.

XML Schema, Part 1 and Part 2 are specifications that explain how schemas are organized in XML documents. The two WS-Security Version 1.0 schemas have been updated to the Version 1.1 specifications plus a new Version 1.1 schema has been added. Note that the Version 1.1 schema does not replace the Version 1.0 schema but instead builds upon it by defining an additional set of capabilities within a Version 1.1 namespace.

You can use the following methods to configure Web services security and to define policy types to secure the SOAP messages:

- **Use the administrative console to configure policy sets.**

  This method uses the bootstrap policy that is defined in the policy set. You can use policy sets, or assertions about how services are defined, to simplify your security configuration for Web services. You can use the administrative console to create, modify, and delete custom policy sets. A set of default policy sets are available.

  For example, you can define the bootstrap policy in the policy set to secure the Web Services Trust (WS-Trust) SOAP messages.

© Copyright IBM Corp. 2008

You can also use the administrative console to perform policy set management tasks and to secure Web Services using encryption, signing information, and security tokens.

The following steps high-level steps describe how to configure WebSphere Application Server to use WS-Security and to secure the SOAP messages using the administrative console. The generator and consumer tasks that are discussed in the following steps use WS-Security Versions 1.0 and 1.1.

- Create and configure the application policy sets or the system policy sets for trust service.
- Define the policy types to be used to secure the SOAP messages when creating and configuring the policy sets.
- Configure the policy set binding. Select either the symmetric or asymmetric binding assertion to describe the token type and the algorithm to be used for message protection.
- Assemble your Web services security-enabled application by using an assembly tool.

- **Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)**

  WebSphere Application Server uses a new API programming model. In addition to the existing JAX-RPC programming model, a new programming model, Java API for XML Web Services (JAX-WS), has been added. The JAX-WS programming standard aligns with the document-centric messaging model and replaces the remote procedure call programming model defined by the Java API for XML-based RPC (JAX-RPC) specification.

  For example, an application could create system policy sets and then use the WebSphere Application Server WSS API to acquire the security context token for programmatic API-based Web Services Secure Conversation (WS-SecureConversation).

  You can also use the administrative console to perform the encryption, signing, and token configuration tasks that the WSS APIs perform to secure Web services.

  The following high-level steps describe how to configure WebSphere Application Server to use WS-Security and to secure the SOAP messages using the WSS APIs. The generator and consumer tasks that are discussed in the following steps use WS-Security Versions 1.0 and 1.1.

  - Use the WSSSignature API to configure the signing information for the request generator (client side) binding.

    Different message parts can be specified in the message protection for a request on the generator side. The default required parts are BODY, ADDRESSING_HEADERS and TIMESTAMP.

    The WSSSignature API also specifies the different algorithm methods to be used with the signature for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.
  - Use the WSSSignPart API if you want to change the digest method and the transform method.

    The default signed parts are WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS and WSSSignature.TIMESTAMP.

    The WSSSignPart API also specifies the different algorithm methods to be used if you added or changed the signed parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N. For example, use the WSSSignPart API if you want to generate the signature for the SOAP message using the SHA256 digest method instead of the default value of SHA1.
  - Use the WSSEncryption API to configure the encryption information on the request generator side.

    The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The default targets of encryption are BODY_CONTENT and SIGNATURE.

    The WSSEncryption API also specifies the different algorithm methods to be used to protect message confidentiality. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.
  - Use the WSSEncryptPart API if you want to set the transform method only.

    For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

No algorithm methods are required for encrypted parts.

– Use the WSS API to configure the token on the generator side.

  The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for creating the security token on the generator side. Different standalone tokens can be sent in request and response. The default token is the X509Token. The other token that can be used for signing is the DerivedKeyToken, which is used only with Web Services Secure Conversation (WS-SecureConversation).

– Use the WSSVerification API to verify the signature for the response consumer (client side) binding.

  Different message parts can be specified in the message protection for a response on the consumer side. The required targets for verification are BODY, ADDRESSING_HEADERS and TIMESTAMP.

  The WSSVerification API also specifies the different algorithm methods to be used for verifying the signature and for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.

– Use the WSSVerifyPart API to change the digest method and the transform method. The required verify parts are WSSVerification.BODY, WSSVerification.ADDRESSING_HEADERS and WSSVerification.TIMESTAMP.

  The WSSVerifyPart API also specifies the different algorithm methods to be used if you added or changed the verification parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N.

– Use the WSSDecryption API to configure the decryption information for the response consumer (client side) binding.

  The decryption information on the consumer side is used for decrypting an incoming SOAP message. The targets of decryption are BODY_CONTENT and SIGNATURE. The default key encryption method is KW_RSA_OAEP.

  No algorithm methods are required for decryption.

– Use the WSSDecryptPart API if you want to set the transform method only.

  For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

  No algorithm methods are required for decrypted parts.

– Use the WSS API to configure the token on the consumer side.

  The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for validating (authenticating) the security token on the consumer side. Different standalone tokens can be sent in request or response.

  The WSS API adds the information for the candidate token that is used for decryption. The default token is X509Token.

- **Use the wsadmin administrative scripting tool to configure policy sets.**

  This method allows you to create, manage, and delete policy sets from the command-line or to create scripts to automate your tasks. You can use the wsadmin tool and the PolicySetManagement command group to manage default policy sets, create custom policy sets, configure policies, and manage attachments and bindings. For more information, use the policy set scripting topics in the information center.

To secure Web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. You must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

## Results

After completing these high-level steps for WebSphere Application Server, you have secured Web services by configuring policy sets and by using the WSS API to configure encryption and decryption, the signature

and signature verification information, and the consumer and generator tokens.

## Securing messages at the request generator using WSS APIs

You can secure SOAP messages by configuring signing information, encryption, and generator tokens to protect message integrity, confidentiality, and authenticity, respectively. This request (client-side) generator configuration defines the Web services security requirements for the outgoing SOAP message request.

### Before you begin

To secure Web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. Therefore, in addition to securing messages at the request generator level, you must also secure messages at the response consumer level.

### About this task

The request (client-side) generator configuration requirements involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches security tokens.

To secure Web service applications, you must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

You can use the following interfaces to configure Web services security and to define policy types to secure the SOAP messages:
- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:
- Configure generator signing to protect message integrity.
- Configure encryption to protect message confidentiality.
- Attach generator tokens to protect message authenticity.

### Results

After completing these procedures, you have secured messages at the request generator level.

### What to do next

Next, if not already configured, secure messages with signature verification, decryption, and consumer tokens at the response consumer (client-side) level.

## Configuring encryption to protect message confidentiality using the WSS APIs

You can configure encryption information for the client-side request generator (sender) bindings. Encryption information is used to specify how the generators (senders) encrypt outgoing SOAP messages. To configure encryption, specify which message parts to encrypt and specify which algorithm methods and security tokens are to be used for encryption.

### Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding 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. Prior to configuring encryption, familiarize yourself with XML encryption.

## About this task

For encryption, you must specify the following:
- Which parts of the message are to be encrypted.
- Which encryption algorithms to specify.

To configure encryption and encrypted parts on the client side, use the WSSEncryption and WSSEncryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses encryption information for the default generator to encrypt parts of the SOAP message. The WSSEncryption API configures the following required parts as encrypted parts.

*Table 8. Required encrypted parts*

| Encryption parts | Description |
| --- | --- |
| Keywords | Keywords are used to add the encrypted parts to the SOAP message. |
| XPath expression | An XPath expression is used to add the encrypted parts to the SOAP message. |
| WSSEncryptPart object | This object adds the encrypted parts to the SOAP message. |
| WSSSignature object | This object adds the signature component as an encrypted part. |
| Header | This part adds the header in the SOAP header, specified by QName, as an encryption part. |
| Security token object | This object adds the security token as an encryption part. |

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the message that are to be encrypted. WebSphere Application Server supports the use of the following keywords:

*Table 9. Supported encryption keywords*

| Keyword | References |
| --- | --- |
| BODY_CONTENT | The keyword for the contents of the SOAP message body as an encryption target. |
| SIGNATURE | The keyword for the signature component as an encryption target. |

If configuring using the WSS APIs, the WSSEncryption and WSSEncryptPart APIs complete these high-level steps:

1. Use the WSSEncryption API to configure encryption. The WSSEncryption API performs these tasks by default:
   a. Generates the callback handler.
   b. Generates the generator security token object.
   c. Adds the security token reference type.
   d. Adds the signature component.

e. Adds the WSSEncryptPart object.
f. Adds the parts to be encrypted. Adds the default parts as targets of encryption by using keywords and XPath expressions.
g. Adds the header in the SOAP message, specified by QName.
h. Sets the default data encryption method.
i. Specifies whether the key is to be encrypted using a Boolean value.
j. Sets the default key encryption method.
k. Selects a part reference.
l. Sets the MTOM optimization Boolean value.

2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the key (false).
   - Change the security token type from default of X.509 token.
   - Change the security token reference type from the default value of SecurityToken.REF_STR.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.
   - Turn MTOM optimization on (true).

## Results

The encryption information is configured for the generator binding.

## Example

The following is an example of the WSSEncryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSGenerationContext gencont = factory.newWSSGenerationContext();

X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);
WSSEncryption enc = factory.newWSSEncryption(token);

gencont.add(enc);
```

## What to do next

You must configure similar decryption information for the client-side response consumer (receiver) bindings, if you have not already configured the information.

Next, review the WSSEncryption API process.

## Encrypting the SOAP message using the WSSEncryption API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for request encryption on the generator side, use the WSSEncryption API to encrypt the SOAP message. The WSSEncryption API specifies which request SOAP message parts to encrypt when configuring the client.

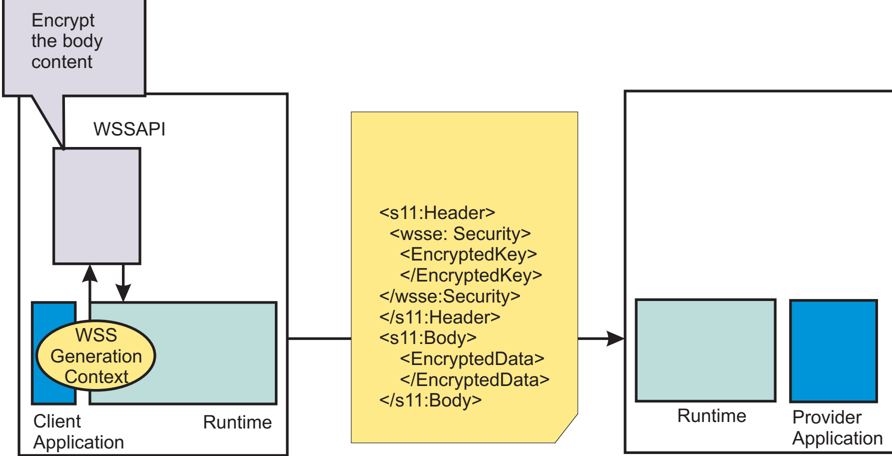
### Before you begin

You can use the WSS API or use policy sets on the administrative console to enable encryption and add generator security tokens in the SOAP message. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts, as needed, using the WSSEncryptPart API.

### About this task

The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The client generator configuration must match the configuration for the provider consumer.




Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted Nonce and timestamp elements to.

The following encryption parts can be configured:

*Table 10. Encryption parts*

| Encryption parts | Description |
|---|---|
| part | Adds the WSSEncryptPart object as a target of the encryption part. |
| keyword | Adds the encryption parts using keywords. WebSphere Application Server supports the following keywords:<br>• BODY_CONTENT<br>• SIGNATURE |
| xpath | Adds the encryption part using an XPath expression. |
| signature | Adds the WSSignature component as a target of the encrypted part. |
| header | Adds the SOAP header, specified by QName, as a target of the encrypted part. |
| securityToken | Adds the SecurityToken object as a target of the encrypted part. |

For encryption, certain default behaviors occur. The simplest way to use the WSSEncryption API is to use the default behavior (see the example code).

WSSEncryption provides defaults for the key encryption algorithm, the data encryption algorithm, the security token reference method, and the encryption parts such as the SOAP body content and the signature. The encryption default behaviors include:

*Table 11. Encryption decisions*

| Encryption decisions | Default behavior |
| --- | --- |
| Which SOAP message parts to encrypt using keywords | Sets the encryption parts that you can add using keywords. The default encryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords:<br>• WSSEncryption.BODY_CONTENT<br>• WSSEncryption.SIGNATURE |
| Which data encryption method to choose (algorithm) | Sets the data encryption method. Both data and key encryption methods can be specified. The default data encryption algorithm method is AES 128. WebSphere Application Server supports these data encryption methods:<br>• WSSEncryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc<br>• WSSEncryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc<br>• WSSEncryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc<br>• WSSEncryption.TRIPLE_DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc |
| Whether to encrypt the key (isEncrypt) | Specifies whether to encrypt the key. The values are `true` or `false`. The default value is to encrypt the key (`true`). |
| Which key encryption method to choose (algorithm) | Sets the key encryption method. Both data and key encryption methods can be specified. The default key encryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods:<br>• WSSEncryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128<br>• WSSEncryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192<br>• WSSEncryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256<br>• WSSEncryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p<br>• WSSEncryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5<br>• WSSEncryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes |
| Which security token to specify (securityToken) | Sets the SecurityToken. The default security token type is the X509Token. WebSphere Application Server provides the following pre-configured consumer token types:<br>• Derived key token<br>• X.509 tokens |
| Which token reference to use (refType) | Sets the type of the security token reference. The default token reference is SecurityToken.REF_KEYID. WebSphere Application Server supports the following token reference types:<br>• SecurityToken.REF_KEYID<br>• SecurityToken.REF_STR<br>• SecurityToken.REF_EMBEDDED<br>• SecurityToken.REF_THUMBPRINT |
| Whether to use MTOM (mtomOptimize) | Sets Message Transmission Optimization Mechanism (MTOM) optimization for the encrypted part. |

1. To encrypt the SOAP message using the WSSEncryption API, first ensure that the application server is installed.
2. The WSS API process for encryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory used for encryption.
   d. Creates WSSEncryption from the WSSFactory instance using the SecurityToken. The default behavior of WSSEncryption is to encrypt the body content and the signature.
   e. Adds a new part to be encrypted in WSSEncryption if the existing part is not appropriate. After addEncryptPart(), addEncryptHeader(), or addEncryptPartByXPath() is called, the default part is cleared.
   f. Calls the encryptKey(false) if the key is not to be encrypted.
   g. Sets the data encryption method if the default method is not appropriate.
   h. Sets the key encryption method if the default method is not appropriate.
   i. Sets the token reference if the default token reference is not appropriate.
   j. Adds WSSEncryption to WSSConsumingContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

## Results

If there is an error condition during encryption, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web services security.

## Example

The following example provides sample code using methods that are defined in WSSEncryption:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance (step: b)
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
X509GenerateCallbackHandler callbackHandler = new
         X509GenerateCallbackHandler(
     "",
     "enc-sender.jceks",
     "jceks",
     "storepass".toCharArray(),
     "bob",
     null,
     "CN=Bob, O=IBM, C=US",
     null);

// Generate the security token used for encryption (step: c)
SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance (step: d)
WSSEncryption enc = factory.newWSSEncryption(token);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
   enc.addEncryptPart(WSSEncryption.BODY_CONTENT);
```

```
// Set the part in the SOAP Header specified by QName (step: e)
   enc.addEncryptHeader(new QName("http://www.w3.org/2005/08/addressing",
                                  "MessageID"));

// Set the part specified by WSSSignature (step: e)
   SecurityToken sigToken = getSecurityToken();
      WSSSignature sig = factory.newWSSSignature(sigToken);
   enc.addEncryptPart(sig);

// Set the part specified by SecurityToken (step: e)
   UNTGenerateCallbackHandler untCallbackHandler =
          new UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
                                             untCallbackHandler);
   enc.addEncryptPart(unt, false);

// sSt the part specified by XPath expression (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
   enc.addEncryptPartByXPath(sb.toString());

// Set whether the key is encrypted (step: f)
// DEFAULT: true
   enc.encryptKey(true);

// Set the data encryption method (step: g)
// DEFAULT: WSSEncryption.AES128
   enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method (step: h)
// DEFAULT: WSSEncryption.KW_RSA_OAEP
   enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Set the token reference (step: i)
// DEFAULT: SecurityToken.REF_KEYID
  enc.setTokenReference(SecurityToken.REF_STR);

// Add the WSSEncryption to the WSSGenerationContext (step: j)
  gencont.add(enc);

// Process the WS-Security header (step: k)
gencont.process(msgcontext);
```

**Note:** The X509GenerationCallbackHandler does not need the key password because the public key is used for encryption. You do not need a password to obtain the public key from the Java keystore.

### What to do next

If you have not previously specified which encryption methods to choose, use the WSS API or configure the policy sets using the administrative console to choose the data and key encryption algorithm methods.

## Choosing the encryption methods for the generator binding

To configure the client for request encryption for the generator binding, you must specify which encryption methods to use when the client encrypts the SOAP messages.

### Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To specify which algorithm methods are to be used when the client encrypts the SOAP messages, complete the following tasks:

- Use the WSSEncryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSEncryptPart API to configure a transform algorithm method, if needed. The default is no transform algorithm.

## About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

**Data encryption method algorithm**

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

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.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

**Key encryption method algorithm**

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the method to encrypt the key that is used to encrypt data. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property.

The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

**Note:** You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

*Table 12. Encryption usage types*

| Usage types | Description |
| --- | --- |
| Data encryption | Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks. |
| Key encryption | Specifies the algorithm URI that is used for encrypting and decrypting the encryption key. |

### Data encryption

WebSphere Application Server supports the following pre-configured data encryption algorithms:

*Table 13. Data encryption algorithms*

| Data encryption name | Algorithm URI |
|---|---|
| WSSEncryption.AES128 (the default value) | A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc |
| WSSEncryption.AES192 | A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc |
| WSSEncryption.AES256 | A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc |
| WSSEncryption.TRIPLE_DES | A URI of data encryption algorithm, 3DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc |

### Key encryption

WebSphere Application Server supports the following pre-configured key encryption algorithms:

*Table 14. Key encryption algorithms*

| Key encryption name | Algorithm URI |
|---|---|
| WSSEncryption.KW_AES128 | A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128 |
| WSSEncryption.KW_AES192 | A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Note:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP). |
| WSSEncryption.KW_AES256 | A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256 |
| WSSEncryption.KW_RSA_OAEP (the default value) | A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p |
| WSSEncryption.KW_RSA15 | A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5 |
| WSSEncryption.KW_TRIPLE_DES | http://www.w3.org/2001/04/xmlenc#kw-tripledes |

To configure the encryption and encrypted part algorithm methods, use the WSSEncryption API, or configure policy sets using the administrative console.

**Note:** Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with Web services endpoints that use policy sets.

The WSS API process completes the following high-level steps to specify which encryption methods to use when configuring the client for request encryption:

1. Using the WSSEncryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The client generator configuration must match the configuration for the provider consumer.

The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data encryption algorithms:

- AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

  The AES 128 algorithm is the default data algorithm method.

- AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

  Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

  To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

- AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

  To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

- TRIPLEDES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
// Default data encryption algorithm: AES128
WSSEncryption enc = factory.newWSSEncryption(x509t);
   enc.setEncryptionMethod(EncryptionMethod.TRIPLEDES_CBC);
gencont.add(enc);
```

3. Using the WSSEncryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. If the encryption key, which is the key that is used for encrypting the message parts, is not encrypted, then the decryption API selects **false** to match the encryption key.

   The client generator configuration must match the configuration for the provider consumer.

   The default key encryption algorithm value is key wrap RSA OAP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

   - KW AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
   - KW AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

     To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).KW AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

     To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

   - KW RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     The KW RSA OAEP algorithm is the default key algorithm method.

     When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p

   - KW RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - KW TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

**Note:** For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
- algorithmName
- keyLength

## Results

If there is an error condition, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web services security.

## Example

The following example provides sample WSS API code using WSSEncryption.setEncryptionMethod() and WSSEncryption.setKeyEncryptionMethod().

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
                                   "",
                                   "enc-sender.jceks",
                                   "jceks",
                                   "storepass".toCharArray(),
                                   "bob",
                                   null,
                                   "CN=Bob, O=IBM, C=US",
                                   null);

// Generate the security token used for encryption
   SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance
   WSSEncryption enc = factory.newWSSEncryption(token);

// Set the data encryption method
// DEFAULT: WSSEncryption.AES128
      enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
      enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Add the WSSEncryption to the WSSGenerationContext
   gencont.add(enc);

// Generate the WS-Security header
gencont.process(msgcontext);
```

## What to do next

Next, if you want to add a transform algorithm, review the WSSEncryptPart API process task.

# Encryption methods

For request generator binding settings, the encryption methods include specifying the data and key encryption algorithms to use to encrypt the SOAP message. The WSS API for encryption (WSSEncryption)

specifies the algorithm name and the matching algorithm uniform resource identifier (URI) for the data and key encryption methods. If the data and key encryption algorithms are specified, only elements that are encrypted with those algorithms are accepted.

## Data encryption algorithms

The data encryption algorithm is used to encrypt parts of the SOAP message, including the body and the signature. Data encryption algorithms specify the algorithm uniform resource identifier (URI) for each type of data encryption algorithms.

The following pre-configured data encryption algorithms are supported:

*Table 15. Data encryption algorithms*

| Data encryption algorithm name | Algorithm URI |
|---|---|
| WSSEncryption.AES128 (the default value) | A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc |
| WSSEncryption.AES192 | A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc |
| WSSEncryption.AES256 | A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc |
| WSSEncryption.TRIPLE_DES | A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc |

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.

For the AES256-cbc and the AES192-CBC algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm configured for encryption for the generator side must match the data encryption algorithm that is configured for decryption for the consumer side.

## Key encryption algorithms

This algorithm is used to encrypt and decrypt keys. This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

**Note:** Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with Web services endpoints using the policy sets.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The following pre-configured key encryption algorithms are supported:

*Table 16. Supported pre-configured key encryption algorithms*

| WSS API | URI |
|---|---|
| WSSEncryption.KW_AES128 | A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128 |

*Table 16. Supported pre-configured key encryption algorithms (continued)*

| WSS API | URI |
| --- | --- |
| WSSEncryption.KW_AES192 | A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Note:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP). |
| WSSEncryption.KW_AES256 | A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256 |
| WSSEncryption.KW_RSA_OAEP (the default value) | A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p |
| WSSEncryption.KW_RSA15 | A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5 |
| WSSEncryption.KW_TRIPLE_DES | A URI of key encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes |

For Secure Conversation, additional key-related information must be specified, such as:

- algorithmName
- keyLength

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

**Note:** You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator must match the key decryption algorithm that is configured for the consumer.

This example provides sample code for encryption to use the Triple DES for the data encryption method and to use RSA1.5 for the key encryption method:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();
```

```
// generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "enc-sender.jceks",
  "jceks",
  "storepass".toCharArray(),
  "bob",
  null,
  "CN=Bob, O=IBM, C=US",
  null);

// generate the security token used to the encryption
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

// generate WSSEncryption instance to encrypt the SOAP body content
WSSEncryption enc = factory.newWSSEncryption(token);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// set the data encryption method
// DEFAULT: WSSEncryption.AES128
enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// add the WSSEncryption to the WSSGenerationContext
gencont.add(enc);

// generate the WS-Security header
gencont.process(msgcontext);
```

## Adding encrypted parts using the WSSEncryptPart API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure encrypted parts for the request generator (client side) bindings, use the WSSEncryptPart API to define and add to the listing of elements in the encrypted part. WSSEncryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.encryption package.

### Before you begin

You can use the WSS APIs or configure policy sets using the administrative console to enable the encrypted parts. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts using the WSSEncryptpart API, as needed.

### About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted elements to. The encryption information on the generator side is used for encrypting an outgoing SOAP message. The request generator is configured for the client.

The WSSEncryptPart API specifies information related to encrypted parts and sets the encrypted parts that have been added for message confidentiality protection. Use the WSSEncryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSEncryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The encrypted parts and related information displayed in the following table are used to protect the confidentiality of messages.

*Table 17. Encrypted parts*

| Encrypted parts | Description |
|---|---|
| part | Adds the WSSEncryptPart object as a target of the encryption part. |
| keyword | Adds the encrypted parts using keywords. The default encryption parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords:<br>• BODY_CONTENT<br>• SIGNATURE |
| xpath | Adds the encrypted part by using an XPath expression. |
| signature | Adds the WSSSignature component as a target of the encrypted part.<br><br>WSSSignature is applicable only if the SOAP message contains a signature element. |
| header | Adds the SOAP header, specified by QName, as a target of the encrypted part. |
| securityToken | Adds the SecurityToken object as a target of the encrypted part. |

For encrypted parts, certain default behaviors occur. The simplest way to use the WSSEncryptPart API is to use the default behavior. The WSSEncryptPart API provides defaults for specifying the transform algorithm, setting objects as targets, specifying the encrypted parts, such as: the SOAP body content and the signature.

The encryption default behaviors include:

*Table 18. Encrypted part decisions*

| Encrypted part decisions | Default behavior |
|---|---|
| Which SOAP message parts to encrypt using keywords | Specifies which keywords to use for the encrypted parts. WebSphere Application Server sets the following SOAP message parts by default for encryption:<br>• WSSEncryption.BODY_CONTENT<br>• WSSEncryption.SIGNATURE |
| Which transform method to add | WebSphere Application Server does not specify any transform method by default. Specify a transform method only if using SOAP with Attachments. |

1. To encrypt the SOAP message parts using the WSSEncryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSEncryptPart follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the encryption.
   d. Creates WSSEncryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSEncryptPart from WSSFactory.
   f. Adds the parts to be encrypted and to be applied with the transform in WSSEncryptPart. WebSphere Application Server sets these encrypted parts by default for WSSEncryptPart: the BODY_CONTENT and SIGNATURE. After you add other encrypted parts, the default values are no longer valid. For example, if you call addEncryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to encrypt the security token, the signature, and the body content, you must call addEncryptPart(securityToken, false),

addEncryptPart(WSSEncryption.SIGNATURE), and addEncryptPart(WSSEncrypyion.BODY_CONTENT).

g. Sets the transform method.
h. Adds WSSEncryptPart to WSSEncryption.
i. Adds WSSEncryption to WSSGenerationContext.
j. Calls WSSGenerationContext.process() with the SOAPMessageContext.

### Results

If there is an error condition during encryption of the message parts, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web services security.

### What to do next

After enabling encrypted parts for the request generator (client side) binding, you must specify the same parts to be decrypted for the response consumer (client side) bindings. Next, to configure decryption and decrypted parts, use the WSS APIs or configure policy sets using the administrative console.

# Configuring generator signing information to protect message integrity using the WSS APIs

You can configure the signing information to protect message integrity for the request (client side) generator binding. Signing information includes the signature and the signed parts. To keep the integrity of the message, digital signatures are typically applied.

## Before you begin

In addition to using a digital signature and configuring the signing information, the following tasks should also be performed:

- Verify the signing information.
- Incorporate encryption.
- Attach security tokens.

## About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using the signature algorithm methods. The WSS APIs specify which algorithm is to be used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports several pre-configured request signing algorithm methods.

You can use the following interfaces to configure Web services security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for the signing information.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client).

Perform the following signing tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the generator binding.

- Configure the signing information using the WSSSignature API. Configure the signing information for the generator binding using the WSSSignature API. Signing information is used to sign parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both signing and encryption can be applied to the same message parts, such as the SOAP body.
- Add or change signed parts using the WSSSignPart API.
- Configure the client for request signing methods using the WSSSignature or WSSSignPart APIs. To configure the client for request signing, choose the signing methods. The request signing methods include the signature, the canonicalization, the digest, and the transform methods. Use the WSSSignature API to configure the signature and canonicalization methods. Use the WSSSignPart API to configure the digest and transform methods.

## Results

The WSS APIs also specify the security token for the generator (client) binding and set the type of token reference to protect message authenticity. By completing the steps in these tasks, you have configured generator signing to protect the integrity of the SOAP message.

## What to do next

Next, verify the consumer signing information by using the WSS APIs or by configuring policy sets using the administrative console.

# Configuring the signing information using the WSS APIs

You can configure the signing information for the client-side request generator (sender) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token. To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

### Before you begin

WebSphere Application Server uses XML digital 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. Prior to completing these steps, familiarize yourself with XML digital signature for signing and verifying digital signatures for digital content.

### About this task

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. WebSphere Application Server uses the signing information for the default generator to sign parts of the message, such as the body, time stamp, and Username token.

For the signing information, you must specify the following:

- Which parts of the message are to be signed.
- The key information that is referenced by the key information for the signing keys.
- The signing algorithms.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSSignature API configures the following parts as signature parts:

*Table 19. Pre-configured signature parts*

| | |
|---|---|
| Security token object | This object authenticates the client. If this option is specified, then the message is signed. You can digitally sign the message using a security token if a login configuration authentication method is selected. |
| WSSTimestamp object | This object adds a time stamp to a message. The time stamp determines if the message is valid based on the time that the message is sent and then received. |
| WSSSignature Part object | This object adds the signature parts to a message. |
| SOAP header and the QName as a target | This signature part adds the header, specified by QName, as a verification part. |

The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be signed. WebSphere Application Server supports the use of the following keywords:

*Table 20. Supported signature keywords*

| Keyword | References |
|---|---|
| ADDRESSING_HEADERS | The Web Services Addressing (WS-Addressing) headers. |
| BODY | The SOAP message body. The body is the user data portion of the message. |
| TIMESTAMP | The creation and expiration timestamp information. |

The Web Services Security API (WSS API) are used to configure the signing information for the request generator (client side) section of the bindings file. To configure the signing information on the client side, use the WSS APIs or configure policy sets for signing using the administrative console.

If configuring using the WSS APIs, the WSSSignature and WSSSignPart APIs complete the following steps to specify which message parts to digitally sign when configuring the client for request generator signing:

1. The WSSSignature API adds the required parts of the SOAP message to digitally sign. Either a keyword or an XPath expression can be used to specify the required encryption parts.
2. The WSSSignature API sets the signature method algorithm. The default signature method is RSA_SHA1. WebSphere Application Server supports the following pre-configured algorithms:
   - RSA SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - HMAC SHA1http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).

   Any ds:SignatureMethod/@Algorithm element in a signature is based on a symmetric key and must have a value of RSA-SHA1 or HMAC-SHA1.

   The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.
3. The WSSSignature API sets the canonicalization method. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured algorithms:
   - The URI of the exclusive canonicalization algorithm, EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#.
   - The URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#.

   The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.
4. The WSSSignature API adds a security token. The API adds information about the security token that is to be used for the signature, such as:

- The class for security token.
- The callback handler
- The name of the JAAS login configuration.

5. The WSSSignature API sets the type of security token and sets the type of token reference. WebSphere Application Server supports the following pre-configured token references:
   - SecurityToken.REF_STR

     Represents the security token reference as a token reference type.
   - SecurityToken.REF_KEYID

     Represents the key identifier reference as a token reference type.
   - SecurityToken.REF_EMBEDDED

     Represents the embedded reference as a token reference type.
   - SecurityToken.REF_THUMBPRINT

     Represents the thumbprint reference as a token reference type.
6. If SecurityToken.REF_KEYID is set as the type of token reference, the WSSSignature API sets the key information signature type and configures the key information that is referenced by the key information references. WebSphere Application Server supports the following:
   - Specifying that the KeyInfo element is not signed.
   - Specifying that the entire <KeyInfo> element is signed.
   - Specifying that the child elements <Keyinfochildelements> of the <KeyInfo> element are signed.

   If you do not specify one of the previous signature types, WebSphere Application Server specifies that the entire <KeyInfo> element is signed, by default.

   If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.

   The key information signature type for the generator must match the signature type for the consumer.
7. The WSSSignature API specifies whether to require signature confirmation. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

   The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.
8. The WSSSignPart API specifies the part reference. The part reference specifies which parts of the message to digitally sign.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature verification: <DigestTransform> and <Transform>.
9. The WSSSignPart API specifies the digest method algorithm. The digest method algorithm specified within the <DigestMethod> element is used in the <SigningInfo> element.

   WebSphere Application Server supports the following pre-configured digest algorithms:
   - http://www.w3.org/2000/09/xmldsig#sha1
   - http://www.w3.org/2001/04/xmlenc#sha256
   - http://www.w3.org/2001/04/xmlenc#sha512
10. The WSSSignPart API specifies the transform algorithm. The transform algorithm is that is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms:
    - http://www.w3.org/2001/10/xml-exc-c14n#

- http://www.w3.org/TR/1999/REC-xpath-19991116

  Do not use this transform algorithm if you want to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
- http://www.w3.org/2002/06/xmldsig-filter2
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
- http://www.w3.org/2002/07/decrypt#XML
- http://www.w3.org/2000/09/xmldsig#enveloped-signature

The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

**Note:** If both of the following conditions are true, WebSphere Application Server signs the referenced token:

- You previously selected the Keyinfo or the Keyinfochildelements option
- You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.

11. If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. Configure policy sets using the administrative console to configure the same actor strings for the Web service on the server, which processes the request and sends the response back.

    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 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.

## Results

After the WSSSignature and WSSSignPart APIs complete these steps, the signing information is configured for the generator sections of the bindings files.

## Example

The following example shows WSS API sample code to configure the signature, to generate the callback handler, and to specify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
// Instantiate a generation context
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler and specify the X.509 token
   X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
   SecurityToken token = factory.newSecurityToken(X509Token.class,
                                                  callbackHandler);

// Set the signature information
WSSSignature sig = factory.newWSSSignature(token);
// Add the header using QName
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "To"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "Action"));
// Apply the signature
```

```
   gencont.add(sig);

// Secure the message
gencont.process(msgctx);
```

### What to do next

You must configure similar signature information for the client-side request consumer (receiver) bindings by completing the following verification tasks:

- Verify the signature
- Choose the signature algorithm methods.
- Change or add signed parts, as needed.

If signature verification is already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

## Configuring the signature information using the WSSSignature API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web services security APIs (WSS API). To configure the signature information for the generator binding sections for the client-side request, use the WSSSignature API. The WSSSignature API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

### Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to enable the signing information. To secure SOAP messages, you must complete the following signature tasks:

- Configure the signature information.
- Choose the signature methods.
- Add or change signed parts, as needed.

### About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-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. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The signing information specifies the integrity constraints that are applied to generated messages. The constraints include specifying which message parts within the generated message must be digitally signed, and the message parts to attach digitally signed Nonce and timestamp elements to. The following signature and related signature part information are configured:

*Table 21. signature parts information*

| signature parts | Description |
|---|---|
| keyword | Adds a signature part using keywords. Use the following keywords for the signature parts:<br>• ADDRESSING_HEADERS<br>• BODY<br>• TIMESTAMP<br>The WS-Addressing headers are not encrypted but can be signed. |
| xpath | Adds a signature part by using an XPath expression. |
| part | Adds a WSSSignPart object as a target of the signature part. |
| timestamp | Adds a WSSTimestamp object as a target of the signature part. When specified, the timestamp information also specifies when the message is generated and when it expires. |
| header | Adds the header, specified by QName, as a target of the signature part. |
| securityToken | Adds a SecurityToken object as a target of the signature part. |

For signing information, certain default behaviors occur. The simplest way to use the WSSSignature API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing method, the canonicalization method, the security token references, and the signature parts.

*Table 22. Signature default behaviors*

| Signature decisions | Default behavior |
|---|---|
| Which keywords to use | Sets the keywords. WebSphere Application Server supports the following keywords by default:<br>• ADDRESSING_HEADERS<br>• BODY<br>• TIMESTAMP |
| Which signature method to use | Sets the signature algorithm. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods:<br>• WSSSignature.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1<br>• WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1<br>The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported. |
| Which canonicalization method to use | Sets the canonicalization algorithm. The default canonicalization method is EXC C14N. WebSphere Application Server supports the following pre-configured canonicalization methods:<br>• WSSSignature.EXC_C14N; http://www.w3.org/2001/10/xml-exc-c14n#<br>• WSSSignature.C14N: http://www.w3.org/2001/10/xml-c14n# |
| Whether signature confirmation is required | Sets whether to require signature confirmation. The default value is **false**. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. If required, the value of your signature confirmation is stored in order to use it to validate the signature confirmation after receiving back the message that generated the signature confirmation in the response message. This method is for the requestor side. |

*Table 22. Signature default behaviors (continued)*

| Signature decisions | Default behavior |
|---|---|
| Which security token to use | Sets the SecurityToken. The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type.<br><br>WebSphere Application Server provides the following pre-configured consumer token types:<br>• Derived Key Token<br>• X509 tokens<br><br>You can also create custom token types, as needed. |
| Which token reference to set | Sets the refType. SecurityToken.REF_STR is the default value for the type of token reference. WebSphere Application Server supports these pre-configured token references types:<br>• SecurityToken.REF_STR<br>• SecurityToken.REF_KEYID<br>• SecurityToken.REF_EMBEDDED<br>• SecurityToken.REF_THUMBPRINT |

If WSSSignature.requireSignatureConfirmation() is called, then the WSSSignature API expects that the response message will include the signature confirmation.

1. To configure the signing information in a SOAP message by using the WSS API, first ensure that the application server is installed.
2. Use the WSSSignature API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance. WSSGenerationContext must be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken. The default behavior of WSSSignature is to sign these signature parts: BODY, ADDRESSING_HEADERS, and TIMESTAMP.
   e. Adds the part to be signed, if the default part is not appropriate. If the digest method or transform method is changed, creates WSSSignPart and add it to WSSSignature.
   f. Creates WSSSignaturePart to WSSSignature. Calls the requiredSignatureConfirmation() method, if the signature confirmation is to be applied.
   g. Sets the canonicalization method, if the default is not appropriate.
   h. Sets the signature method, if the default is not appropriate.
   i. Sets the token reference, if the default is not appropriate.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

## Results

You have completed the steps to configure the signature for the generator section of the bindings. If there is an error condition when signing the message parts, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

## Example

The following example provides sample code that uses methods that are defined in the WSSignature API.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance  (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
       "",
       "dsig-sender.ks",
       "jks",
       "client".toCharArray(),
       "soaprequester",
       "client".toCharArray(),
       "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token to be used for the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class,
        callbackHandler);

// Generate the WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part to be signed  (step: e)
// DEFAULT: WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS,
//          and WSSSignature.TIMESTAMP.

// Set the part in the SOAP Header specified by QName  (step: e)
      sig.addSignHeader(new
                        QName("http://www.w3.org/2005/08/addressing",
                        "MessageID"));

// Set the part specified by the keyword  (step: e)
      sig.addSignPart(WSSSignature.BODY);

// Set the part specified by SecurityToken  (step: e)
   UNTGenerateCallbackHandler untCallbackHandler = new
      UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
         untCallbackHandler);
      sig.addSignPart(unt);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();
      sigPart.setSignPart(WSSSignature.TIMESTAMP);
      sigPart.setDigestMethod(WSSSignPart.SHA256);
      sig.addSignPart(sigPart);

// Set the part specified by WSSTimestamp  (step: e)
   WSSTimestamp timestamp = factory.newWSSTimestamp();
      sig.addSignPart(timestamp);

// Set the part specified by XPath expression  (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
      sig.addSignPartByXPath(sb.toString());
```

```
// Set to apply the signature confirmation  (step: f)
      sig.requireSignatureConfirmation();

// Set the canonicalization method  (step: g)
// DEFAULT: WSSSignature.EXC_C14N
      sig.setCanonicalizationMethod(WSSSignature.C14N);

// Set the signature method  (step: h)
// DEFAULT: WSSSignature.RSA_SHA1
      sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

// Set the token reference  (step: i)
// DEFAULT: SecurityToken.REF_STR
      sig.setTokenReference(SecurityToken.REF_KEYID);

// Add the WSSSignature to WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgctx);
```

**Note:** The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

### What to do next

Next, chose the algorithm methods if you want a method that is different from the default values. If the algorithm methods do not need to be changed, next use the WSSVerification API to verify the signature and specify the algorithm methods in the consumer section of the binding. Note that the WSSVerification API is only supported on the response consumer (client side).

## Adding signed parts using the WSSSignPart API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure parts to be signed for the request generator (client side) bindings, use the WSSSignPart API to protect the integrity of messages and to configure the digest and transform algorithm methods. The WSSSignPart API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

### Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to configure the signing information. To secure SOAP messages using the signing information, you must complete one of the following tasks:

- Configure the signature information
- Configure signed parts, as needed.

### About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing digest and transform algorithms (for example, SHA1 or TRANSFORM_EXC_C14N).

The signing information specifies the integrity constraints that are applied to generated messages. The signed parts are used to protect the integrity of messages. You can specify the signed parts to add for message integrity protection.

The following table shows the required signed parts when the digital signature security constraint (integrity) is defined:

*Table 23. Signed parts information*

| Signed parts | Description |
| --- | --- |
| keyword | Adds signed parts using keywords. WebSphere Application Server supports the following keywords for signed parts:<br>• BODY<br>• ADDRESSING_HEADERS<br>• TIMESTAMP<br><br>The WS-Addressing headers are not encrypted but can be signed. |
| xpath | Adds the required signed parts by using an XPath expression. |
| header | Adds the header, specified by QName, as a signed part. |
| timestamp | Adds a WSSTimestamp object as a signed part. If specified, the timestamp information specifies when the message is generated and when it expires. |

Different message parts can be specified in the message protection for request on the generator side. WSSSignPart allows for adding a transform algorithm, setting a digest method, setting objects as targets, specifying whether an element, and the signed parts, such as: the SOAP body, the WS-Addressing header, and timestamp information.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignPart API is to use the default behavior (see the example code). The signed parts default behaviors include:

| Signature decisions | Default behavior |
| --- | --- |
| Which SOAP message parts to sign | WebSphere Application Server supports the following SOAP message parts to be signed and used for message protection:<br>• WSSSignature.BODY<br>• WSSSignature.ADDRESSING_HEADERS<br>• WSSSignature.TIMESTAMP |
| Which digest method to use | Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element.<br><br>WebSphere Application Server supports the following pre-configured digest methods:<br>• WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1<br>• WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256<br>• WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512 |

| Signature decisions | Default behavior |
|---|---|
| Which transform algorithms to use | Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature.<br><br>WebSphere Application Server supports the following pre-configured transform algorithms:<br>• WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#<br>• WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2<br>Use this transform method to ensure compliance with the Basic Security Profile (BSP).<br>• WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform<br>• WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature |

1. To enable Web Service Security by using the WSS API (WSSSignPart), first ensure that the application server is installed.
2. Use the WSSSignPart API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signed parts follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken.
   e. Creates WSSSignPart from the WSSFactory instance.
   f. Sets the part to be signed and the digest method or transform method specified by step g or step h if the default is not appropriate.
   g. Sets the digest method if the default is not appropriate.
   h. Sets the transform method if the default is not appropriate.
   i. Adds WSSSignPart to WSSSignature. After any WSSSignPart is set to WSSSignature, the default parts to be signed, which are specified in WSSSignature, are ignored.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

## Results

You have completed the steps to configure the signed parts for the generator section of the bindings files. If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web services security is applied to the SOAP message.

## Example

The following example provides sample code that uses all of methods that are defined in the WSSSignPart API:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance  (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
```

```
// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler
      "",
      "dsig-sender.ks",
      "jks",
      "client".toCharArray(),
      "soaprequester",
      "client".toCharArray(),
      "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token used to the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

// Generate WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();

// Set the part specified by WSSSignPart  (step: f)
   sigPart.setSignPart(WSSSignature.BODY);

// Set the digest method specified by WSSSignPart  (step: g)
   sigPart.setDigestMethod(WSSSignPart.SHA256);

// Set the transform method specified by WSSSignPart  (step: h)
   sigPart.setTransformMethod(WSSSignPart.TRANSFORM_STRT10);

// Add the part specified by WSSSignPart  (step: i)
   sig.addSignPart(sigPart);

// Add the WSSSignature to the WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgcontext);
```

**Note:** The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

### What to do next

Use the WSSVerifyPart API or configure policy sets using the administrative console to verify the signed parts on the consumer side.

## Configuring the client for request signing methods

Use the WSSSignature and WSSSignPart APIs to choose the signing methods. The request signing methods include the signature, canonicalization, digest, and transform methods.

### Before you begin

First, you must have specified which parts of the message sent by the client must be digitally signed using the WSS APIs or configuring policy sets using the administrative console.

### About this task

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.

*Table 24. Signing methods*

| Name of method | Description |
| --- | --- |
| Canonicalization algorithm | Canonicalizes the `<SignedInfo>` element before the information is digested as part of the signature operation. |
| Signature algorithm | Calculates the signature value of the canonicalized `<SignedInfo>` element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration. |
| Transform method | Transforms the parts to be signed before the information is digested as part of the signature operation. |
| Digest method | Calculates the digest value of the transformed parts. The algorithm selected for the client request sender configuration must match the algorithms selected in the server request receiver configuration. |

You can use the WSS APIs or configure policy sets using the administrative console to configure the signing algorithm methods. If using the WSS APIs, use the WSSSignature and WSSSignPart APIs to specify which message parts to digitally sign when configuring the client for request signing.

The WSSSignature and WSSSignPart APIs complete the following steps to configure the signature and signed part algorithm methods:

1. For the generator binding, the WSSSignature API specifies the signature method. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSSignature.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 digital signature method, http://www.w3.org/2000/09/xmldsig#dsa-sha1.
2. For the generator binding, the WSSSignature API specifies the canonicalization method. WebSphere Application Server supports the following pre-configured canonicalization algorithms:
   - WSSSignature.EXC_C14N (the default value): The exclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignature.C14N: The inclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-c14n#
3. For the generator binding, the WSSSignPart API specifies the digest method. WebSphere Application Server supports the following pre-configured digest methods:
   - WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the generator binding, the WSSSignPart API specifies the transform method. WebSphere Application Server supports the following pre-configured transform algorithms:
   - WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:
   - http://www.w3.org/TR/1999/REC-xpath-19991116
   - http://www.w3.org/2002/07/decrypt#XML

## Results

Using the WSS APIs, you have specified which algorithm methods are used to digitally sign a message when the client sends a message to a server.

## Example

The following example is sample code for specifying the signature information, HMAC_SHA1 as signature method, C14N as a canonicalizaion method, SHA256 as a digest method, and EXC_C14N and TRANSFORM_STRT10 as the transform methods:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the digest method
 // DEFAULT: WSSSignPart.SHA1
sigPart.setDigestMethod(WSSSignPart.SHA256);

//add the transform method
// DEFAULT: WSSSignPart.TRANSFORM_EXC_C14N
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_EXC_C14N);
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_STRT10);

 // add the WSSSignPart to the WSSSignature
 sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

### What to do next

After you configure the client to digitally sign the message and to choose the algorithm methods, you must configure the server to verify the digital signature for request signing and to choose the algorithm methods.

Configure policy sets using the administrative console to configure the signature verification information and methods on the server.

## Digital signing methods using the WSSSignature API

You can configure the signing information for the generator binding using the WSS API. To configure the client for request signing, choose the digital signing methods. The algorithm methods include the signing and canonicalization methods.

You must configure generator signing information to protect message integrity by digitally signing SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message.

### Methods

Methods that are used for the signing information include the:

**Signature method**
Sets the signature algorithm method.

**Canonicalization method**
Sets the canonicalization algorithm method.

### Signature algorithms

The signature algorithms specify the algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports the following pre-configured algorithms:

*Table 25. Signature algorithms*

| Algorithm | Description |
|---|---|
| WSSSignature.HMAC_SHA1 | A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1 |
| WSSSignature.RSA_SHA1 (the default value) | A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1 |

For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 algorithm, http://www.w3.org/2000/09/xmldsig#dsa-sha1

The signing algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

### Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

*Table 26. Signature canonicalization algorithms*

| Algorithm | Description |
|---|---|
| WSSSignature.EXC_C14N (the default value) | A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n# |
| WSSSignature.C14N | A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n# |

The canonicalization algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

The following example provides sample WSS API code that specifies the HMAC_SHA1 as a signature method and C14n as a canonicalization method:

```
//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
     X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

## Signed parts methods using the WSSSignPart API

You can configure the signed parts information for the generator binding using the WSS API. The algorithms include the digest and transform methods.

You can protect message integrity by configuring signed parts and key information. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

### Methods

Methods that are used for the signed parts include the:

**Digest method**
Sets the digest algorithm method.

**Transform algorithm**
Sets the transform algorithm method.

## Digest algorithms

The digest method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

*Table 27. Signed parts digest methods*

| Digest method | Description |
|---|---|
| WSSSignPart.SHA1 (the default value) | A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1 |
| WSSSignPart.SHA256 | A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256 |
| WSSSignPart.SHA512 | A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512 |

## Transform algorithms

The transform method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

*Table 28. Signed parts transform methods*

| Digest method | Description |
|---|---|
| WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE | A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature |
| WSSSignPart.TRANSFORM_STRT10 | A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform |
| WSSSignPart.TRANSFORM_EXC_C14N (the default value) | A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n# |
| WSSSignPart.TRANSFORM_XPATH2_FILTER | A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2 |

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part.

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code for specifying the signature and signed parts, setting the signing key and adding the STR-Transform transform algorithm as signed parts:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();
```

```
//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the part specified by WSSSignPart
sigPart.setSignPart(WSSSignature.BODY);

//set the digest method specified by WSSSignPart
sigPart.setDigestMethod(WSSSignPart.SHA256);

//set the transform method specified by WSSSignPart
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);

//set the part specified by WSSSignPart
sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

# Attaching the generator token using WSS APIs to protect message authenticity

When you specify the token generator, the information is used on the generator side to generate the security token.

## Before you begin

The token processing and pluggable token architecture in the Web Service Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Service Security run time.

**Note:** The com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent interface is not used with JAX-WS Web services. If you are using JAX-RPC Web services, this interface is still valid.

Note that the key name (KeyName) element is not supported in the application server because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification. For similar reasons, a SAML token is not supported out of the box.

## About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web services security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web services security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Service Security run time.

**Note:** WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation
- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the generator side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching consumer security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the generator-side security token, use the appropriate pre-configured token generator interface from the WSS APIs to complete the following token configuration process steps:

1. Generate the wssFactory instance.

2. Generate the wssGenerationContext instance.

   The WSSGenerationContext interface stores the components for generating Web Services Security (WS-Security), such as the signing and encryption information, the security token, and the time stamp. When the generate() method is called, all of these components are generated.
3. Create the generator-side components, such as the WSSSignature and the WSSEncryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.
5. Specifiy a token generator class name. The token generator class name specifies the required information to generate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specifiy the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web services security header within the SOAP message. Also, the token generator is a plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the WSSGenerationContext interface.

   WebSphere Application Server provides the following default callback handler implementations for the generator side:

   **com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback**
   : This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   **com.ibm.websphere.wssecurity.callbackhandler.UNTGUIPromptCallbackHandler**
   : This class is a callback handler for the Username token with the GUI prompt on the generator side. This instance is used to set the WSSGenerationContext object to generate a Username token.

   **com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler**
   : This class is a callback handler for the Username token on the generator side. This instance is used to set into WSSGenerationContext object to attach a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   **com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler**
   : This class is a callback handler that is used to generate the X.509 certificate that is inserted in the Web services security header within the SOAP message as a binary security token on the generator side. This instance is used to generate the WSSSignature and WSSEncryption objects, set the objects into the WSSGenerationContext object to generate the X.509 binary security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must be provided on the generator side.

   **com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler**
   : This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the generator side. This instance is used to generate WSSSignature object and WSSEncryption object to generate a LTPA token.

     This callback handler is used to validate the LTPA security token inserted in the Web services security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run

As Subject. Use this callback handler only when the Web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

**com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler**
This class is a callback handler for the Kerberos v5 token on the generator side. This instance is used to set the WSSGenerationContext object to generate the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSSignature and WSSEncryption objects to use the Kerberos session key or derived key in the SOAP message signature and encryption.

7. If a X.509 token is specified, additional token information is also specified.

| | |
|---|---|
| storeRef | The reference name of the keystore. |
| storePath | The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations. |
| storePassword | The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server. |
| storeType | The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection:<br>**JKS** Use this option if the keystore uses the Java Keystore (JKS) format.<br>**JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption.<br>**JCERACFKS**<br>Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS® only).<br>**PKCS11KS (PKCS11)**<br>Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.<br>**PKCS12KS (PKCS12)**<br>Use this option if your keystore uses the PKCS#12 file format. |
| alias | The key alias name. The key alias is used by the key locator to find the key within the keystore file. |
| keyPassword | The key password that is used for recovering the key. This password is needed to access the key object within the keystore file. |
| keyName | The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`. |
| certStores | A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). This step configures a collection certificate store and certificate revocation lists for the generator bindings. |

| identityAssertion | Specifies whether identity assertion is used. Selects this item if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web services security header within the SOAP message. For an X.509 token generator, the application server sends the original signer certification only. |
|---|---|
| requestorCertificate | Specifies whether the certificate of the requestor is used. |

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain.

d. With keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

**ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509**
Specifies an X.509 certificate token.

**ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1**
Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web services security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

**ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7**
Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web services security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

**ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2**
For an LTPA token, you can use `LTPA` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

**ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2**
For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes`http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token generator class name, the following token information can be specified:
   a. Whether to use IdentityAssertion option. This option is selected if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web services security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a Username token generator.
   b. Whether to use RunAsSubject identity option. This option is used if an identity assertion is defined and you want to use the Run As identity instead of the initial caller identity for identity assertion in a downstream call. This option is valid only if you have configured the Username token as the token generator.
   c. Whether to use sendRealm.
   d. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the generated token type is a Username token, and it is available only for the request generator binding.
   e. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   f. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

      **URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken**
      Specifies a Username token.
9. If the Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

| Token Information | Description | Default Value |
| --- | --- | --- |
| name | Kerberos client principal name | |
| password | Kerberos client password | |
| realm | Kerberos realm associated with the Kerberos client | Default realm name in Kerberos configuration file.<br><br>Specify null to use the default value. |
| targetService | Kerberos service name associated with the target Web Services. | |
| targetHost | Kerberos realm name associated with the Kerberos service name. | |
| tokenValueType | Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification. | http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ |
| targetRealm | Kerberos realm name associated with the Kerberos service name. | Default realm name in the Kerberos configuration file |
| prompt | A boolean value to enable the login prompt. | false |

| Token Information | Description | Default Value |
|---|---|---|
| supportTokenRequireSHA1 | A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token. | false<br><br>SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed. |
| alwaysAPREQ | A boolean value to indicate that the client should always send the Kerberos AP_REQ token in the request messages. | false<br><br>The SHA1 key is used instead in the subsequent messages. If set to true, the Kerberos AP_REQ token is always used. |
| requireDKT | A boolean value to require a derived key for message protection. | false |
| clabel | The client label for the derived key. | WS-SecureConversation<br><br>Specify null to use the default value. |
| slabel | The service label for the derived key. | WS-SecureConversation<br><br>Specify null to use the default value. |
| keylen | The length of the derived key. | 16<br><br>Specify zero to use the default value |
| noncelen | The length of the nonce. | 16<br><br>Specify zero to use the default value |
| encComponent | An instance of WSSEncryption. | Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component. |
| sigComponent | An instance of WSSSignature. | Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component. |

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If Secure Conversation is used for message protection, the following information must be specified:

| Information | Description |
| --- | --- |
| bootstrapWSSGenerationContext | The bootstrap configuration used to secure the RequestSecurityToken (RST) token. |
| bootstrapWSSConmingContext | The bootstrap configuration used for consuming a secured RequestSecurityTokenResponse (RSTR). |
| ENDPOINT_URL | The service end point URL. |
| EncryptionAlgorithm | This determines the key size. |
| cLabel | The client label used when creating the derived key. |
| sLabel | The server label used when creating the derived key. |

11. Set the components into the wssGenerationContext object.
12. Invoke the wssGenerationContext.process() method.

## Results

Using the Web Services Security API (WSS API) process, you can configured the token generator.

## What to do next

Next, you must specify a similar token consumer configuration.

# Configuring the generator security tokens using the WSS API

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the generator side, use the Web Services Security APIs (WSS API). The generator security tokens are part of the com.ibm.websphere.wssecurity.wssapi.token interface package.

## Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security runtime and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must complete the following token tasks:

- Configure the generator tokens.
- Configure the consumer tokens.

## About this task

The JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token on the generator side.

On the generator side, the token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Service Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Service Security runtime to the LoginModule. After the token is authenticated, a security token object is created, and the token is passed it to the Web Service Security runtime.

When using the WSS API for generator token creation, certain default behaviors occur. The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide default values for the token type, the token value, and the JAAS confirmation name. The default token behaviors include:

| Generator token decisions | Default behavior |
| --- | --- |
| Which token type to use | The token type specifies which type of token to use for message integrity, message confidentiality, or message authenticity.<br><br>WebSphere Application Server provides the following pre-configured generator token types for message integrity and message confidentiality:<br>• Derived key token<br>• X509 tokens<br><br>You can also create custom token types, as needed.<br><br>WebSphere Application Server also provides the following pre-configured generator token types for the message authenticity:<br>• Username token<br>• LTPA tokens<br>• X509 tokens<br><br>You can also create custom token types, as needed. |
| What JAAS login configuration name to specify | The JAAS login configuration name specifies which JAAS login configuration name to use. |
| Which configuration type to use | The JAAS login module specifies the configuration type. Only the pre-configured generator configuration types can be used for generator token types. |

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, the XML format, and the cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following tokens types are subclasses of the generic security token class:

*Table 29. Subclasses of the SecurityToken*

| Token type | JAAS login configuration name |
| --- | --- |
| Username token | system.wss.generate.unt |
| Security context token | system.wss.generate.sct |
| Derived key token | system.wss.generate.dkt |

The following tokens types are subclasses of the binary security token class:

*Table 30. Subclasses of the BinarySecurityToken*

| Token type | JAAS login configuration name |
| --- | --- |
| LTPA token | system.wss.generate.ltpa |
| LTPA propagation token | system.wss.generate.ltpaProp |
| X.509 token | system.wss.generate.x509 |
| X.509 PKI Path token | system.wss.generate.pkiPath |
| X.509 PKCS7 token | system.wss.generate.pkcs7 |

**Notes®:**

- For each JAAS login token generator configuration name, there is a respective token consumer configuration name. For example, for the Username token, the respective token consumer configuration name is system.wss.consume.unt.
- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

1. To configure the securityToken package, com.ibm.websphere.wssecurity.wssapi.token, first ensure that the application server is installed.
2. Use the Web Services Security token generator process to configure the tokens. For each token type, the process is similar to the following process that demonstrates the UsernameToken token generator process:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates a JAAS CallbackHandler. The authentication data, such as the user name and password are specified as part of the CallbackHandler. For example, the following code specifies Chris as the user name and sirhC as the password: UNTGenerationCallbackHandler("Chris", "sirhC");
   d. Calls any JAAS CallbackHandler parameters and reviews the token class information for which parameters are required or optional. For example, for the UsernameToken, the following parameters can be configured also:

      **Nonce**
      Indicates whether a nonce is included in the user name token for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The nonce value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

      **Created timestamp**
      Indicates whether to insert a time stamp into the UsernameToken. The timestamp value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.
   e. Creates the SecurityToken from WSSFactory.

      By default, the UsernameToken API specifies the ValueType as: "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"

      By default, the UsernameToken API provides the QName of this class and specifies the NamespaceURI as http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd and also specifies the LocalPart as UsernameToken.
   f. Optional: Specifies the JAAS login module configuration name. On the generator side, the configuration type is always `generate` (for example, system.wss.generate.unt).
   g. Adds the SecurityToken to the WSSGenerationContext.
   h. Calls WSSGenerationContext.process() and generates the WS-Security header.

## Results

If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext process() is called, and the security token for the generator binding is attached.

## Example

The following example code shows how the WSS API process creates a Username security token, attaches the Username token to the SOAP message, and configures the Username token in the generator binding.

```
// get the message context
   Object msgcontext = getMessageContext();

// generate WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// generate WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// generate callback handler
   UNTGenerateCallbackHandler untCallbackHandler =
 new UNTGenerateCallbackHandler("Chris", "sirhC");

// generate the username token
   SecurityToken unt = factory.newSecurityToken(UsernameToken.class, untCallbackHandler);

// add the SecurityToken to the WSSGenerationContext
   gencont.add(unt);

// generate the WS-Security header
gencont.process(msgcontext);
```

The following example shows how to use secure conversation with the WSS APIs to configure the generator tokens, as well as the consumer tokens. In this example, the SecurityContextToken token is created using the WS-SecureConversation draft namespace: `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`. To use the WS-SecureConversation version 1.3 namespace, `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`, specify SecurityContextToken13.class instead of SecurityContextToken.class.

```
WSSGenerationContext bootstrapGenCon =
                           wssFactory.newWSSGenerationContext();

  // Create a Timestamp
                ..
  //add Timestamp
                ..


  // Sign the SOAP Body, WS-Addressing headers, and Timestamp
  X509GenerateCallbackHandler btspReqSigCbHandler = new X509GenerateCallbackHandler(
    ...);
  SecurityToken btspReqSigToken = wssFactory.newSecurityToken(
    X509Token.class, btspReqSigCbHandler);
  WSSSignature bootstrapReqSig = wssFactory.newWSSSignature(btspReqSigToken);
  bootstrapReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
                //add Sign Parts
  ..
                bootstrapGenCon.add(bootstrapReqSig);


  // Encrypt the SOAP Body and the Signature
  X509GenerateCallbackHandler btspReqEncCbHandler = new X509GenerateCallbackHandler(
    ...);
  SecurityToken btspReqEncToken = wssFactory.newSecurityToken(X509Token.class, btspReqEncCbHandler);
  WSSEncryption bootstrapReqEnc = wssFactory.newWSSEncryption(btspReqEncToken);
  bootstrapReqEnc.setEncryptionMethod(WSSEncryption.AES128);
  bootstrapReqEnc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);
                // add Encryption parts
  ..
                ..
                bootstrapGenCon.add(bootstrapReqEnc);

                WSSConsumingContext bootstrapConCon = wssFactory.newWSSConsumingContext();
                X509ConsumeCallbackHandler btspRspVfyCbHandler = new X509ConsumeCallbackHandler(....);
                WSSVerification bootstrapRspVfy = wssFactory.newWSSVerification(X509Token.class, btspRspVfyCbHandler);
                bootstrapRspVfy.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

                //add Verify parts
                ..
                ..
                bootstrapConCon.add(bootstrapRspVfy);

                X509ConsumeCallbackHandler btspRspDecCbHandler = new X509ConsumeCallbackHandler(...);
                WSSDecryption bootstrapRspDec = wssFactory.newWSSDecryption(X509Token.class, btspRspDecCbHandler);
                bootstrapRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
  bootstrapRspDec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA15);
                // add Decryption parts
                ..
                ..
                bootstrapConCon.add(bootstrapRspDec);
```

```
            SCTGenerateCallbackHandler sctgch =
 new SCTGenerateCallbackHandler(bootstrapGenCon, bootstrapConCon,
   ENDPOINT_URL, WSSEncryption.AES128);

            SecurityToken[] scts = wssFactory.newSecurityTokens(new Class[]{SecurityContextToken.class}, sctgch);
            SecurityContextToken sct = (SecurityContextToken)scts[0];


             // Use the SCT to generate DKTs for Secure Conversation
             //Signature algorithm and client and service labels
DerivedKeyToken dktSig = sct.getDerivedKeyToken(WSSSignature.HMAC_SHA1, "WS-SecureConversation", "WS-SecureConversation");
             //Encryption algorithm and client and service labels
DerivedKeyToken dktEnc = sct.getDerivedKeyToken(WSSEncryption.AES128, "WS-SecureConversation", "WS-SecureConversation");

            // Create the application generation context for the request message
WSSGenerationContext applicationGenCon = wssFactory.newWSSGenerationContext();
// Create and add Timestamp
..
..
// add the derived key token and Sign the SOAP Body and WS-Addressing headers
WSSSignature appReqSig = wssFactory.newWSSSignature(dktSig);

appReqSig.setSignatureMethod(WSSSignature.HMAC_SHA1);
appReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
..
            ..
applicationGenCon.add(appReqSig);
// add the derived key token and Encrypt the SOAP Body and the Signature
WSSEncryption appReqEnc = wssFactory.newWSSEncryption(dktEnc);

appReqEnc.setEncryptionMethod(WSSEncryption.AES128);
appReqEnc.setTokenReference(SecurityToken.REF_STR);
appReqEnc.encryptKey(false);
..
            ..
applicationGenCon.add(appReqEnc);

            // Create the application consuming context for the response message
WSSConsumingContext applicationConCon = wssFactory.newWSSConsumingContext();
            //client and service labels and decryption algorithm
SCTConsumeCallbackHandler sctCbHandler =
            new SCTConsumeCallbackHandler("WS-SecureConversation", "WS-SecureConversation", WSSDecryption.AES128);
// Derive the token from SCT and use it to Decrypt the SOAP Body and the Signature
WSSDecryption appRspDec = wssFactory.newWSSDecryption(SecurityContextToken.class, sctCbHandler);
appRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
appRspDec.encryptKey(false);

            ..
applicationConCon.add(appRspDec);
//Derived the token from SCT and use it to Verify the signature on the SOAP Body, WS-Addressing headers, and Timestamp
WSSVerification appRspVfy = wssFactory.newWSSVerification(SecurityContextToken.class, sctCbHandler);
..
            ..
applicationConCon.add(appRspVfy);

            ..
            applicationGenCon.process(messageContext);
applicationConCon.process(messageContext);
```

## What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar consumer tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by signing the SOAP message or by encrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

# Securing messages at the response consumer using WSS APIs

You can secure SOAP messages with signature verification, decryption, and consumer tokens to protect message integrity, confidentiality, and authenticity, respectively. The response consumer (client-side) configuration defines the Web services security requirements for the incoming SOAP response.

## About this task

To secure Web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. You must specify several different configurations. Although there is no

specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

The response consumer (client-side) configuration requirements involve verifying that the integrity parts are signed and that the signature is verified, verifying that the required confidential parts are encrypted and that the parts are decrypted; and validating the security tokens.

You can use the following methods to configure Web services security and to define policy types to secure the SOAP messages:
- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:
- Verify consumer signing information to protect message integrity.
- Configure decryption to protect message confidentiality.
- Validate consumer tokens to protect message authenticity.

### Results

After completing these procedures, you have secured messages at the response consumer level.

### What to do next

Next, if not already configured, secure messages with signing information, encryption, and generator tokens at the response (client-side) generator level.

## Configuring decryption to protect message confidentiality using the WSS APIs

You can configure decryption information for the response consumer (client side) section of the binding file. Decryption information is used to specify how the consumers (receivers) decrypt incoming SOAP messages. To configure decryption, specify which message parts to decrypt and specify which algorithm methods and security tokens are to be used for decryption.

### Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding 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. Prior to configuring decryption, familiarize yourself with XML encryption.

### About this task

For decryption, you must specify the following:
- Which parts of the message are to be decrypted.
- Which decryption algorithms to specify.

To configure decryption and decrypted parts on the client side, use the WSSDecryption and WSSDecryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses decryption information for the default consumer to decrypt parts of the SOAP message. The WSSDecryption API configures the following required parts as decrypted parts.

*Table 31. Required decrypted parts*

| Decryption parts | Description |
| --- | --- |
| Keywords | Keywords are used to add the decrypted parts to the SOAP message. |
| XPath expression | XPath expressions are used to add the decrypted parts to the SOAP message. |
| WSSDencryptPart object | This object adds the decrypted parts to the SOAP message. |
| WSSVerification object | This object adds the signature verification component as a decrypted part. |
| Header | This part adds the header in the SOAP header, specified by QName, as a decrypted part. |
| Security token object | This object adds the security token as a decrypted part. |

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the SOAP message that are to be decrypted. WebSphere Application Server supports the use of the following keywords:

*Table 32. Supported decryption keywords*

| Keyword | References |
| --- | --- |
| BODY_CONTENT | The keyword for the body contents of the SOAP message body as a decryption target. |
| SIGNATURE | The keyword for the signature element as a decryption target. |
| USERNAME_TOKEN, | The keyword for the Username token element as a decryption target. |

If configuring using the WSS APIs, the WSSDecryption and WSSDecryptPart APIs complete these high-level steps:

1. Use the WSSDecryption API to configure encryption. The WSSDecryption API performs these tasks by default:
   a. Generates the callback handler.
   b. Generates the consumer security token object.
   c. Adds the security token reference type.
   d. Adds the WSSEncryptPart object.
   e. Adds the parts to be encrypted. Adds the default parts for decryption by using keywords and XPath expressions.
   f. Adds the verification component.
   g. Adds the header in the SOAP message, specified by QName.
   h. Sets the default data encryption method.
   i. Specifies whether the key is to be decrypted using a Boolean value. Calls this method when the shared key is encrypted.
   j. Sets the default key encryption method.
2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.

b. Sets the encrypted parts specified by an XPath expression.
c. Sets the signature component object, WSSSignature.
d. Sets the header in the SOAP message, specified by QName.
e. Sets the generator security token.
f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Add USERNAME_TOKEN as a target of decryption.
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the encryption key (false).
   - Change the security token type from the default value of X.509 token.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.

## Results

The decryption information is configured for the consumer binding.

## Example

The following is an example of the WSSDecryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();
    X509ConsumeCallbackHandler callbackhandler = generateCallbackHandler();
// see X509ConsumeCallbackHandler
    WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
        callbackhandler);

 concont.add(dec);
```

## What to do next

You must configure similar encryption information for the client-side request generator (sender) bindings, if you have not already configured the information.

Next, review the WSSDecryption API process.

# Decrypting the SOAP message using the WSSDecryption API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for decryption on the response (client) consumer side, use the WSSDecryption API to decrypt the SOAP messages. The WSSDecryption API specifies which request SOAP message parts to decrypt when configuring the client.

### Before you begin

You can use the WSS API or use policy sets on the administrative console to enable decryption and add consumer security tokens in the SOAP message. To secure SOAP messages, you must have completed the following decryption tasks:

- Encrypted the SOAP message.
- Chosen the decryption method.

## About this task

The decryption information on the consumer side is used for decrypting an incoming SOAP message for the response consumer (client side) bindings. The client consumer configuration must match the configuration for the provider generator.

Confidentiality settings require that confidentiality constraints be applied to generated messages.

The following decryption parts can be configured:

*Table 33. Decryption parts*

| Decryption parts | Description |
| --- | --- |
| part | Adds the WSSDecryptPart object as a target of the decryption part. |
| keyword | Adds the decryption part using keywords. WebSphere Application Server supports the following keywords:<br>• BODY_CONTENT<br>• SIGNATURE<br>• USERNAME_TOKEN |
| xpath | Adds the decryption part using an XPath expression. |
| verification | Adds the WSSVerification instance as a target of the decryption part. |
| header | Adds the SOAP header, specified by QName, as a target of the decryption part. |

For decryption, certain default behaviors occur. The simplest way to use the WSS API for decryption is to use the default behavior (see the example code). WSSDecryption provides defaults for the key encryption algorithm, the data encryption algorithm, and the decryption parts such as the SOAP body content and the signature. The decryption default behaviors include:

*Table 34. Decryption decisions*

| Decryption decisions | Default behavior |
| --- | --- |
| Which parts to decrypt | The default decryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords:<br>• WSSDecryption.BODY_CONTENT<br>• WSSDecryption.SIGNATURE<br>• WSSDecryption.USERNAME_TOKEN<br><br>After you specify which message parts to decrypt, you must specify which method to use when decrypting the consumer request message. For example, if both signature and body content are applied for encryption, then the SOAP message parts that are decrypted include the same parts. |
| Whether to encrypt the key (isEncrypt) | The default value is to encrypt the key (`true`). |
| Which data decryption algorithm to choose (method) | The default data decryption algorithm method is AES128. WebSphere Application Server supports these data encryption methods:<br>• WSSDecryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc<br>• WSSDecryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc<br>• WSSDecryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc<br>• WSSDecryption.TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc |

Table 34. Decryption decisions (continued)

| Decryption decisions | Default behavior |
|---|---|
| Which key decryption method to choose (algorithm) | The default key decryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods:<br>• WSSDecryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128<br>• WSSDecryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192<br>• WSSDecryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256<br>• WSSDecryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p<br>• WSSDecryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5<br>• WSSDecryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes |
| Which security token to specify | The default security token type is the X509 token. WebSphere Application Server provides the following pre-configured consumer token types:<br>• Derived key token<br>• X509 tokens |

1. To decrypt the SOAP message using the WSSDecryption API, first ensure that the application server is installed.
2. The WSS API process for decryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. The WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates the callback handler for the consumer side.
   d. Creates WSSDecryption with the class for the security token and the callback handler from the WSSFactory instance. The default behavior of WSSDecryption is to assume that the body content and the signature are encrypted.
   e. Adds the parts to be decrypted, if the default is not appropriate.
   f. Adds the candidates of the data encryption methods to use for decryption.
   g. Adds the candidates of the key encryption methods to use for decryption.
   h. Adds the candidates of the security token to use for decryption.
   i. Calls WSSDecryption.encryptKey(false) if the application does not want the key to be encrypted in the incoming message.
   j. Adds WSSDecryption to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAPMessageContext

## Results

If there is an error condition during decryption, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web services security is applied to the SOAP message.

## Example

The following example provides sample code for decrypting the SOAP message body content:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();
```

```
// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
       X509ConsumeCallbackHandler(
                                "",
                                "enc-sender.jceks",
                                "jceks",
                                "storepass".toCharArray(),
                                "alice",
                                "keypass".toCharArray(),
 "CN=Alice, O=IBM, C=US");

// Generate the WSSDecryption instance (step: d)
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                              callbackHandler);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
      dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
    dec.addRequiredDecryptHeader(new
       QName("http://www.w3.org/2005/08/addressing",
       "MessageID"));

// Set the part specified by WSSVerification  (step: e)
   X509ConsumeCallbackHandler verifyCallbackHandler =
      getCallbackHandler();
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                  verifyCallbackHandler);
      dec.addRequiredDecryptPart(ver);

// Set the part specified by XPath expression (step: e)
      StringBuffer sb = new StringBuffer();
         sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Envelope']");
         sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Body']");
         sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Ping']");
         sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Text']");
      dec.addRequiredDecryptPartByXPath(sb.toString());

// Set the part in the SOAP header to be decrypted specified by QName (step: e)
      dec.addRequiredDecryptHeader(new
          QName("http://www.w3.org/2005/08/addressing",
          "MessageID"));

// Set the candidates for the data encryption method (step: f)
// DEFAULT : WSSDecryption.AES128
      dec.addAllowedEncryptionMethod(WSSDecryption.AES128);
      dec.addAllowedEncryptionMethod(WSSDecryption.AES192);

// Set the candidates for the key encryption method (step: g)
// DEFAULT : WSSDecryption.KW_RSA_OAEP
      dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);

// Set the candidate security token to used for the decryption (step: h)
   X509ConsumeCallbackHandler callbackHandler2 = getCallbackHandler2();
      dec.addToken(X509Token.class, callbackHandler2);

// Set whether or not the key should be encrypted in the incoming SOAP message (step: i)
// DEFAULT: true
```

```
      dec.encryptKey(true);

// Add the WSSDecryption to the WSSConsumingContext (step: j)
   concont.add(dec);

// Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

### What to do next

Next, use the WSSDecryptPart API or configure the policy sets using the administrative console to add decrypted parts for the consumer message.

## Choosing the decryption methods for the consumer binding

To configure the client for response decryption for the consumer binding, specify which data and transform algorithm methods to use when the client decrypts the SOAP messages.

### Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To complete decryption configuration to secure SOAP messages, you must complete the following tasks:

- Configure decryption of the SOAP message parts
- Specify the decryption methods.

You can configure the decryption methods using the WSSDecryption and WSSDecryptPart APIs. Or you can also configure policy sets using the administrative console to configure the decryption methods.

### About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

**Data encryption method algorithm**

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

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.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

**Key encryption method algorithm**

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA_OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA_OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

**Note:** You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW_AES256 and the KW_AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

To complete the decryption configuration, you must specify the algorithm uniform resource identifier (URI) and its usage type. If the URI is used for multiple usage types, then you must define the URI to each usage type. WebSphere Application Server supports the following decryption usage types:

*Table 35. Decryption usage types*

| Usage types | Description |
| --- | --- |
| Data encryption | Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks. |
| Key encryption | Specifies the algorithm URI that is used for encrypting and decrypting the encryption key. |

To configure the decryption and decrypted part algorithms, use the WSSDecryption and WSSDecryptPart APIs, or configure policy sets using the administrative console.

**Note:** Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with Web services endpoints that use policy sets.

If you are using the WSS APIs, the WSSDecryption and WSSDecryptPart APIs specify which algorithm methods are used when the client decrypts the SOAP messages.

- Use the WSSDecryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSDecryptPart API to configure a transform algorithm method.

The WSS API process completes the following high-level steps to specify which decryption and decrypted part algorithm methods to use when configuring the client for response decryption:

1. Using the WSSDecryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data decryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data decryption algorithms:

   - AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

- TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
dec.addAllowedKeyEncryptionMethod(WSSDecryption.TRIPLE_DES);
```

3. Using the WSSDecryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. If no key for encrypting the data is needed, then you must specify WSSDecryption.encryptKey(false).

   The key encryption algorithm that you select for the consumer side must match the key encryption method that you select for the generator side.

   The default key encryption algorithm value is key wrap RSA_OAEP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

   - KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
   - KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

     To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     **Note:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
   - KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256

     To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     The KW_RSA_OAEP algorithm is the default key algorithm method.

     When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
   - KW_RSA_15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

   **Note:** For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
   - algorithmName
   - keyLength

4. As needed, uses the WSSDecryption API method to change to other key encryption algorithms. For example, you might add the following code to change from the default key encryption algorithm KW_RSA_OAEP to the TRIPLE_DES algorithm:

```
dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);
```

5. Using the WSSDecryptPart API, adds a transform algorithm, as needed. There is no default transform algorithm. However, WebSphere Application Server provides a pre-configured decrypted part, WSSDecryptPart.TRANSFORM_ATTACHMENT_CIPHERTEXT, that can be added.

### Results

If there is an error condition, a WSSException is provided. If successful, the API calls the WSSConsumerContext.process(), the WS-Security header is validated, and the SOAP message is now secured using Web services security.

### Example

The following example provides sample WSS API code for decrypting the body content as well as changing the data encryption and key encryption algorithms from the default values:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
                                 "",
                                 "enc-sender.jceks",
                                 "jceks",
                                 "storepass".toCharArray(),
                                 "alice",
                                 "keypass".toCharArray(),
                                 "CN=Alice, O=IBM, C=US");

// Generate WSSDecryption instance
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                             callbackHandler);

// Set the candidates for the data encryption method
// DEFAULT : WSSDecryption.AES128
   dec.addAllowedEncryptionMethod(WSSDecryption.AES128);
   dec.addAllowedEncryptionMethod(WSSDecryption.AES192);

// Set the candidates for the key encryption method
// DEFAULT : WSSDecryption.KW_RSA_OAEP
   dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);

// Add the WSSDecryption to WSSConsumingContext
   concont.add(dec);

// Validate the WS-Security header
concont.process(msgcontext);
```

## Adding decrypted parts using the WSSDecryptPart API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web services security APIs (WSS API). To configure decrypted parts for the response consumer (client side) bindings, use the WSSDecryptPart API to define and add to the listing of elements in the decrypted part. WSSDecryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.decryption package.

### Before you begin

You can use either the WSS APIs or configure the policy sets using the administrative console to configure and add new encrypted parts. To secure SOAP messages using the WSSDecryptPart APIs, you must

configure the decrypted parts for the response consumer bindings.

## About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted and decrypted, and which message parts to attach encrypted elements to.

The WSSDecryptPart API specifies information related to decryption and sets the decrypted parts that have been added for message confidentiality protection. Use the WSSDecryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSDecryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The decrypted parts displayed in the following table are used to protect the confidentiality of messages.

*Table 36. Decrypted Parts*

| **Decrypted parts** | **Description** |
| --- | --- |
| keyword | Sets the decrypted part using keywords. The default decrypted parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports the following keywords:<br>• BODY_CONTENT<br>• SIGNATURE<br>• USERNAME_TOKEN |
| xpath | Sets the decrypted part by using an XPath expression. |
| verification | Sets the WSSVerification component as a decrypted part.<br><br>The WSSVerification part is applicable only if the SOAP message contains a signature element. |
| header | Sets the header, specified by QName, as a decrypted part. |

For decrypted parts, certain default behaviors occur. The simplest way to use the WSSDecryptPart API is to use the default behavior (see the example code).

WSSDecryptPart provides defaults for setting the transform algorithm, adding a transform method, setting objects as targets, whether an element, and the encrypted parts, such as: the SOAP body content and the signature.

*Table 37. Decrypted part decisions*

| **Decryption decisions** | **Default behavior** |
| --- | --- |
| Which SOAP message parts to decrypt using keywords | Specifies which keywords to use for the decrypted parts. WebSphere Application Server sets the following SOAP message parts by default for decryption:<br>• WSSDecryption.BODY_CONTENT<br>• WSSDecryption.SIGNATURE |
| Which transform algorithm to use (algorithm) | WebSphere Application Server does not specify any transform algorithm by default. Specify a transform method only if using SOAP with Attachments. |

1. To decrypt the SOAP message parts using the WSSDecryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSDecryptPart follows these steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.

b. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
c. Creates the SecurityToken from WSSFactory to configure decryption.
d. Creates WSSDecryption from the WSSFactory instance using SecurityToken.
e. Creates WSSDecryptPart from the WSSFactory instance. The default behavior of WSSDecryptPart is to assume that the body content and signature are encrypted.
f. Adds the parts to be decrypted and to be applied with the transform in WSSDecryptPart. WebSphere Application Server sets these encrypted parts by default for WSSDecryptPart: the BODY_CONTENT and SIGNATURE. After you add other decrypted parts, the default values are no longer valid. For example, if you call addDecryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to decrypt the security token, the signature, and the body content, you must call addDecryptPart(securityToken, false), addDecryptPart(WSSDecryption.SIGNATURE), and addDecryptPart(WSSDecryption.BODY_CONTENT).
g. Sets the transform method.
h. Adds WSSDecryptPart to WSSDecryption.
i. Adds WSSDecryption to WSSConsumingContext.
j. Calls WSSConsumingContext.process() with the SOAPMessageContext

### Results

If there is an error condition when decrypting the message, a WSSException is provided. If successful, the API calls the WSSConsumingContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web services security.

### What to do next

After enabling decrypted parts for the response consumer (client side) binding, specify the generator and consumer tokens, if the security tokens have not already been specified.

## Decryption methods

The decryption algorithms specify the data and key encryption algorithms that are used to decrypt the SOAP message. The WSS API for decryption (WSSDecryption) specifies the algorithm uniform resource identifier (URI) of the data and key encryption methods. The WSSDecryption interface is part of the com.ibm.websphere.wssecurity.wssapi.decryption package.

### Data encryption algorithms

The data encryption algorithms are the algorithms that are used to encrypt and decrypt data. This algorithm type is used for encrypting data to encrypt and decrypt various parts of the message, including the body content and the signature.

Data decryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. WebSphere Application Server supports the following pre-configured data decryption algorithms:

*Table 38. Supported pre-configured data decryption algorithms*

| WSS API | URI |
| --- | --- |
| WSSDecryption.AES128 (the default value) | A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc |
| WSSDecryption.AES192 | A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc |
| WSSDecryption.AES256 | A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc |

Table 38. Supported pre-configured data decryption algorithms (continued)

| WSS API | URI |
| --- | --- |
| WSSDecryption.TRIPLE_DES | A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc |

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.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm must match the data decryption algorithm that is configured for the consumer.

## Key encryption algorithms

The key encryption algorithms are the algorithms that are used to encrypt and decrypt keys.

This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 39. Supported pre-configured key encryption algorithms

| WSS API | URI |
| --- | --- |
| WSSDecryption.KW_AES128 | A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128 |
| WSSDecryption.KW_AES192 | A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192<br>**Note:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP). |
| WSSDecryption.KW_AES256 | A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256 |
| WSSDecryption.KW_RSA_OAEP (the default value) | A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p |
| WSSDecryption.KW_RSA15 | A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5 |
| WSSDecryption.KW_TRIPLE_DES | A URI of data encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes |

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method: http://www.w3.org/2001/04/xmlenc#sha256 http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

**Note:** You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the kw-aes256 and the kw-aes192 key encryption algorithms, you must download the unrestricted JCE policy files from the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator and the consumer must match.

The following example provides a sample of the WSS API code for the default algorithms that are used for WebSphere Application Server decryption:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();

// Required to attach username token into the message.
X509ConsumeCallbackHandler callbackHandler =
       new X509ConsumeCallbackHandler("",
                                      "enc-sender.jceks",
                                      "JCEKS",
                                      "storepass".toCharArray(),
                                      "alice",
                                      "keypass".toCharArray(),
                                      "CN=Alice, O=IBM, C=US");
// Set the decrypt component.
// Default encrypted part: Body-Content
// Default data encryption algorithm: AES128
// Default key encryption algorithm: KW-RSA-OAEP
WSSDecryption dec = factory.newWSSDecryption(X509Token.Type,
callbackHandler);
concont.add(dec);

// validate the WS-Security header.
concont.process(msgctx);
```

## Verifying consumer signing information to protect message integrity using WSS APIs

You can verify the signing information to protect message integrity for the response (client side) consumer binding. Signing information includes the signature and the signed parts for the generator side as well as signature verification and verify parts for the consumer side. To keep the integrity of the message, digital signatures are typically applied.

### Before you begin

Ensure that the signature and signed parts information has been configured. The signature verification information must match what was configured on the generator side.

### About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by verifying the digitally signed body, time stamp, and WS-Addressing headers using the signature verification algorithm methods. The WSS APIs specify which algorithm is to be

used to verify the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature verification method. WebSphere Application Server supports several pre-configured verification algorithm methods.

You can use the following interfaces to configure Web services security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for signature verification.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

Perform the following verification tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the consumer binding.

- Configure the signing information using the WSSSignature API. Configure the signature verification information for the consumer binding using the WSSVerification API. Signature verification information is used to verify parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both verifying and decryption can be applied to the same message parts, such as the SOAP body.
- Add or change verify parts using the WSSVerifyPart API.
- Configure the client for request signing methods using the WSSVerification or WSSVerifyPart APIs. To configure the client for response verification, choose the verification methods. Use the WSSVerification API to configure the canonicalization and signature methods. Use the WSSVerifyPart API to configure the digest and transform methods.

## Results

By completing the steps in these tasks, you have configured the consumer verification information to protect the integrity of messages.

## Verifying the signature information for the consumer binding using the WSS APIs

You can configure the signing information for the client-side response consumer (receiver) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token.

### Before you begin

WebSphere Application Server uses XML digital 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. Prior to completing these steps, read the information about XML digital signature to become familiar with signing and verifying digital signatures for digital content.

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet.

Before you can verify the signature and SOAP message signed parts, you must have completed the following tasks:

- Configured the signature.
- Added signed parts, as needed.
- Chosen the signature and signed parts methods.

## About this task

Use the Web Services Security APIs (WSS API) to configure the signing verification information for the response consumer (client side) section of the bindings file. Use the WSSVerification or WSSVerifyPart APIs to configure the client for request signature verification and to specify which digitally signed message parts to verify.

WebSphere Application Server uses the signing information on the consumer side to verify the integrity of the received SOAP message by validating that the message parts (such as the body, time stamp, and Username token) are signed.

On the client side, use the WSS APIs, or configure policy sets using the administrative console to specify which parts of the message are signed and to configure the key information that is referenced by the key information references. To verify the signature and signed parts, use the WSSVerification and WSSVerifyPart APIs.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSVerification and WSSVerifyPart APIs complete the following steps to specify which digitally signed message parts to verify when configuring the client for response consumer signing:

1. The WSSVerification API adds the required verify parts of the SOAP message.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature: <DigestTransform> and <Transform>.

   The WSSVerification API configures the following parts as verification parts:

| | |
|---|---|
| Security token | Adds information for the security token that is used for the signature verification. |
| SOAP header and the QName as a target | Adds the SOAP header, specified by QName, as a verification part. |

   The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be verified. WebSphere Application Server supports the use of the following keywords:

| Keyword | References |
|---|---|
| WSSVerification.ADDRESSING_HEADERS | The Web Services Addressing (WS-Addressing) headers. |
| WSSVerification.BODY | The SOAP message body. The body is the user data portion of the message. |
| WSSVerification.TIMESTAMP | The creation and expiration timestamp information. |

2. The WSSVerification API adds the required header to the SOAP message. The header, specified by QName, is a required verification header.
3. The WSSVerification API adds a security token. Adds information about the security token that is to be used for the signature verification, such as:
   - The class for security token.
   - The callback handler
   - The name of the JAAS login configuration.
4. The WSSVerification API adds the signature method algorithm. The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> element in the binding file into the <SignatureValue> element. The algorithm that is specified for the consumer, which is the response

consumer configuration, must match the algorithm specified for the request generator configuration. WebSphere Application Server supports the following pre-configured signature algorithms:

- WSSVerification.RSA_SHA1:http://www.w3.org/2000/09/xmldsig#rsa-sha1
- WSSVerification.HMAC_SHA1:http://www.w3.org/2000/09/xmldsig#hmac-sha1

WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).

5. The WSSVerification API adds a canonicalization method. The canonicalization method algorithm is used to canonicalize the <SignedInfo> element before it is incorporated as part of the digital signature operation. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.

   WebSphere Application Server supports the following pre-configured canonicalization algorithms:

   - WSSVerification.EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerification.C14N: http://www.w3.org/TR/xml-c14n
6. The WSSVerification API verifies whether a signature confirmation is required. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

   The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.
7. The WSSVerifyPart API adds a digest method. For each part reference in the signing information, the API specifies both a digest method algorithm and a transform algorithm.

   WebSphere Application Server supports the following pre-configured digest algorithms:

   - WSSVerifyPart.SHA1: http://www.w3.org/2000/09/xmldsig#sha1
   - WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
8. The WSSVerifyPart API adds a transform method. For each part reference in the signing information, the API specifies both a digest method algorithm and a transform algorithm.

   WebSphere Application Server supports the following pre-configured transform algorithms:

   - WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support these algorithms:

   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/TR/1999/REC-xpath-19991116

   The transform algorithm for the consumer must match the transform algorithm for the generator.

## Results

You have completed the steps to configure the signing information for the client-side response consumer sections of the bindings files.

## Example

The following example shows WSS API sample code to verify the signature and to verify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();
// Generate the X.509 Callback Handler on the consumer side
    X509ConsumeCallbackHandler callbackhandler = generateCallbackHandler();
    WSSVerification ver = factory.newWSSVerification(X509Token.class,
        callbackhandler);
concont.add(ver);
```

### What to do next

If not already configured, specify a similar signing information configuration for the generator bindings.

Next, if already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

## Verifying the signature using the WSSVerification API

You can secure the SOAP messages, without using policy sets for configuration, by using the Web services security APIs (WSS API). To verify the signing information for the consumer binding sections for the client side request, use the WSSVerification API. You must also specify which algorithm methods and which signature parts of the SOAP message are to be verified. The WSSVerification API is part of the com.ibm.websphere.wssecurity.wssapi.verification package.

### Before you begin

Use the WSS APIs, or configure the policy sets by using the administrative console to verify the signing information. To secure SOAP messages, you must complete the following signature tasks:
- Configure the signature information.
- Choose the algorithm methods for signature and signature verification.
- Verify the signature information.

### About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-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. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined.

*Table 40. Signature verification parts*

| Verification parts | Description |
| --- | --- |
| keywords | Adds required signature parts as targets of verification by using keywords . Different message parts can be specified in the message protection for request on the generator side. Use the following keywords for the required signature verification parts:<br>• ADDRESSING_HEADERS<br>• BODY<br>• TIMESTAMP<br>The WS-Addressing headers are not encrypted but can be signed. |
| xpath | Adds verification parts by using an XPath expression. |

Table 40. Signature verification parts (continued)

| Verification parts | Description |
| --- | --- |
| part | Adds the WSSVerifyPart object as a verification part. |
| header | Adds the header, specified by QName, as a verification part. |

For signature verification information, certain default behaviors occur. The simplest way to use the WSSVerification API is to use the default behavior.

The default values are defined by the WSS API for the digest method, the transform method, the security token, and the required verification parts.

Table 41. Signature verification default behaviors

| Signature verification decisions | Default behavior |
| --- | --- |
| Which signature method to use (algorithm) | Sets the signature algorithm method. Both the data encryption and the signature and the canonicalization can be specified. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods:<br>• WSSVerification.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1<br>• WSSVerification.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1<br>The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported. |
| Which canonicalization method to use (algorithm) | Sets the canonicalization algorithm method. Both the data encryption and the signature and the canonicalization can be specified. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured canonicalization methods:<br>• WSSVerification.EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#<br>• WSSVerification.C14N: http://www.w3.org/2001/10/xml-c14n# |
| Whether signature confirmation is required | If the WSSSignature API specifies that signature confirmation is required, then the WSSVerification API verifies the signature confirmation value in the response message that has the signature confirmation value attached to it when received. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification.<br>The default signature confirmation is false. |
| Which security token to specify (securityToken) | Adds the securityToken object as a signature part. WebSphere Application Server sets the token information to use for verification.<br>WebSphere Application Server supports the following pre-configured tokens for signing:<br>• X.509 Token<br>• Derived Key Token<br>Information required for tokens include the class for the token, the callback handler information, and the name of the JAAS login module. |

1. To verify the signature in a SOAP message by using the WSSVerification API, first ensure that the application server is installed.
2. Use the WSSVerification API to set the message parts to be verified and to specify the algorithms in a SOAP message. The WSS API process for signature verification follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance.

c. Ensures that WSSConsumingContext is called in the JAX-WS Provider implementation class. Due to the nature of the JAX-WS programming model, a JAX-WS provider needs to be implemented and must call the WSSConsumingContext to verify the SOAP message signature.
d. Creates WSSVerification from the WSSFactory instance.
e. Adds the part to be verified. If the digest method or the transform method are changed, create WSSVerifyPart and set it into WSSVerification.
f. Sets the candidates of the canonicalization method, if the default is not appropriate.
g. Sets the candidates of the signature method, if the default is not appropriate.
h. Sets the candidate security token, if the default is not appropriate.
i. Calls the requireSignatureConfirmation(), if the signature confirmation is applied.
j. Adds WSSVerification to WSSConsumingContext.
k. Calls WSSConsumingContext.process() with the SOAP message context.

## Results

You have completed the steps to verify the signature for the consumer section of the bindings. If there is an error condition, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is applied to the SOAP message.

## Example

The following example provides sample code that uses methods that are defined in the WSSVerification API:

```
// Get the message context
  Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext concont = factory.newWSSConsumingContext();


// Generate the certificate list
  String certpath = "c:/WebSphere/AppServer/etc/ws-security/samples/intca2.cer";
// The location of the X509 certificate file
  X509Certificate x509cert = null;
     try {
      InputStream is = new FileInputStream(certpath);
      CertificateFactory cf = CertificateFactory.getInstance("X.509");
      x509cert = (X509Certificate)cf.generateCertificate(is);
       } catch(FileNotFoundException e1){
        throw new WSSException(e1);
       } catch (CertificateException e2) {
        throw new WSSException(e2);
       }
      Set<Object> eeCerts = new HashSet<Object>();
       eeCerts.add(x509cert);
// Create the certificate store
   java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
   CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);
   CertStore cert = null;
   try {
    cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
   } catch (NoSuchProviderException e1) {
    throw new WSSException(e1);
   } catch (InvalidAlgorithmParameterException e2) {
    throw new WSSException(e2);
   } catch (NoSuchAlgorithmException e3) {
    throw new WSSException (e3);
   }
   if(certList != null ){
```

```
    certList.add(cert);
   }
// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new X509ConsumeCallbackHandler(
     "dsig-receiver.ks",
     "jks",
     "server".toCharArray(),
     certList,
     java.security.Security.getProvider("IBMCertPath")
   );

// Generate the WSSVerification instance (step: d)
  WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandler);

// Set the part to be verified (step: e)
// DEFAULT: WSSVerification.BODY, WSSSignature.ADDRESSING_HEADERS,
// and WSSSignature.TIMESTAMP.

// Set the part in the SOAP header to be specified by QName (step: e)
     ver.addRequiredVerifyHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));

// Set the part to be specified by the keyword (step: e)
     ver.addRequiredVerifyPart(WSSVerification.BODY);

// Set the part to be specified by WSSVerifyPart (step: e)
  WSSVerifyPart verPart = factory.newWSSVerifyPart();
     verPart.setRequiredVerifyPart(WSSVerification.BODY);
     verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);
     ver.addRequiredVerifyPart(verPart);

// Set the part specified by XPath expression (step: e)
  StringBuffer sb = new StringBuffer();
        sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Envelope']");
        sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Body']");
        sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Ping']");
        sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Text']");
     ver.addRequiredVerifyPartByXPath(sb.toString());


// Set one or more canonicalization method candidates for verification (step: f)
// DEFAULT : WSSVerification.EXC_C14N
    ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
    ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Set one or more signature method candidates for verification (step: g)
// DEFAULT : WSSVerification.RSA_SHA1
    ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

// Set the candidate security token to used for the verification (step: h)
  X509ConsumeCallbackHandler callbackHandler2 = getCallbackHandler2();
    ver.addToken(X509Token.class, callbackHandler2);


// Set the flag to require the signature confirmation (step: i)
   ver.requireSignatureConfirmation();

// Add the WSSVerification to the WSSConsumingContext (step: j)
  concont.add(ver);

//Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

### What to do next

After verifying the signature and setting algorithm methods for the SOAP message, you can set either the digest method or the transform method. If you want to set these methods, use the WSSVerifyPart API, or

configure policy sets using the administrative console.

## Verifying the signed parts using the WSSVerifyPart API

To secure SOAP messages on the consumer side, use the Web Services Security APIs (WSS API) to configure the verify parts information for the consumer binding on the response consumer (client side). You can specify which algorithm methods and which parts of the SOAP message are to be verified. Use the WSSVerifyPart API to change the digest method or the transform method. The WSSVerifyPart API is part of the com.ibm.websphere.wssecurity.wssapi.verification package.

### Before you begin

To secure SOAP messages using the signing verification information, you must complete one of the following tasks:

- Configure the signature verification information using the WSSVerification API.
- Configure verify parts using the WSSVerifyPart API, as needed.

The WSSVerifyPart is used for specify the transform or digest methods for the verification. Use the WSSVerifyPart API or configure policy sets using the administrative console.

### About this task

WebSphere Application Server uses the signing information for the default consumer to verify the signed parts of the message. The WSSVerifyPart API is only supported on the response consumer (requester).

The following table shows the required verification parts when the digital signature security constraint (integrity) is defined:

*Table 42. Verify parts information*

| Verify parts information | Description |
| --- | --- |
| keyword | Sets the verify parts using the following keywords:<br>• BODY<br>• ADDRESSING_HEADERS<br>• TIMESTAMP<br><br>The WS-Addressing headers are not decrypted but can be signed and verified. |
| xpath | Sets the verify parts using an XPath expression. |
| header | Sets the header, specified by QName, as a required verify part. |

For signature verification, certain default behaviors occur. The simplest way to use the WSSVerification API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing algorithm and the canonicalization algorithm, and the verify parts.

*Table 43. Verify parts default behaviors*

| Verify parts decisions | Default behavior |
| --- | --- |
| Which keywords to specify | The different SOAP message parts to be signed and used for message protection. WebSphere Application Server supports the following keywords:<br>• WSSVerification.BODY<br>• WSSVerification.ADDRESSING_HEADERS<br>• WSSVerification.TIMESTAMP |

*Table 43. Verify parts default behaviors (continued)*

| Verify parts decisions | Default behavior |
| --- | --- |
| Which transform method to use (algorithm) | Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. The default transform method is TRANSFORM_EXC_C14N.<br><br>WebSphere Application Server supports the following pre-configured transform algorithms:<br>• WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#<br>• WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2<br>Use this transform method to ensure compliance with the Basic Security Profile (BSP).<br>• WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform<br>• WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature |
| Which digest method to use (algorithm) | Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. The default digest method is SHA1.<br><br>WebSphere Application Server supports the following digest method algorithms:<br>• WSSVerifyPart.SHA1: http://www.w3.org/2000/09/xmldsig#sha1<br>• WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256<br>• WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512 |

1. To verify signed parts by using the WSSVerifyPart API, first ensure that the application server is installed.
2. Use the Web Services Security API to verify the verification in a SOAP message. The WSS API process for verifying the signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Ensures that WSSConsumingContext is called in the JAX-WS Provider implementation class. Due to the nature of the JAX-WS programming model, a JAX-WS provider needs to be implemented and must call the WSSConsumingContext to verify the SOAP message signature.
   c. Creates the CallbackHandler to use for verification.
   d. Create the WSSVerification object from the WSSFactory instance.
   e. Creates WSSVerifyPart from the WSSFactory instance.
   f. Sets the part to be verified, if the default is not appropriate.
   g. Sets the candidates for the digest method, if the default is not appropriate.
   h. Sets the candidates for the transform method, if the default is not appropriate.
   i. Adds WSSVerifyPart to WSSVerification.
   j. Adds WSSVerification to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAPMessageContext.

## Results

You have completed the steps to verify to verify the signed parts on the consumer side. If there is an error condition when verifying the signing information, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is verified for the SOAP message.

## Example

The following example provides sample code for the WSSVerification API process for verifying the signing information in a SOAP message:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext concont = factory.newWSSConsumingContext();

// Generate the certificate list
   String certpath =
   "c:/WebSphere/AppServer/etc/ws-security/samples/intca2.cer";
// The location of the X509 certificate file
   X509Certificate x509cert = null;
      try {
          InputStream is = new FileInputStream(certpath);
          CertificateFactory cf = CertificateFactory.getInstance("X.509");
          x509cert = (X509Certificate)cf.generateCertificate(is);
      } catch(FileNotFoundException e1){
            throw new WSSException(e1);
      } catch (CertificateException e2) {
            throw new WSSException(e2);
      }

      Set<Object> eeCerts = new HashSet<Object>();
      eeCerts.add(x509cert);
// create certStore
      java.util.List<CertStore> certList = new
          java.util.ArrayList<CertStore>();
      CollectionCertStoreParameters certparam = new
          CollectionCertStoreParameters(eeCerts);
      CertStore cert = null;
      try {
         cert = CertStore.getInstance("Collection",
              certparam, "IBMCertPath");
      } catch (NoSuchProviderException e1) {
           throw new WSSException(e1);
      } catch (InvalidAlgorithmParameterException e2) {
           throw new WSSException(e2);
      } catch (NoSuchAlgorithmException e3) {
           throw new WSSException (e3);
      }
      if(certList != null ){
      certList.add(cert);
      }

// generate callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
       X509ConsumeCallbackHandler(
                                 "dsig-receiver.ks",
                                 "jks",
                                 "server".toCharArray(),
                                 certList,
       java.security.Security.getProvider("IBMCertPath")
);

// Generate the WSSVerification instance (step: d)
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
       callbackHandler);

// Set the part to be specified by WSSVerifyPart (step: e)
   WSSVerifyPart verPart = factory.newWSSVerifyPart();

// Set the part to be specified by the keyword (step: f)
   verPart.setRequiredVerifyPart(WSSVerification.BODY);
```

```
// Set the candidates for the digest method for verification (step: g)
// DEFAULT : WSSVerifyPart.SHA1
   verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

// Set the candidates for the transform method for verification (step: h)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N : String
   verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

// Set WSSVerifyPart to WSSVerification (step: i)
   ver.addRequiredVerifyPart(verPart);

// Add WSSVerification to WSSConsumingContext (step: j)
   concont.add(ver);

//Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

### What to do next

You have completed configuring the signed part to be verified.

## Configuring the client for response signature verification methods

Use the WSSVerification and WSSVerifyPart APIs to choose the signing verification methods. The request signing verification methods include the digest algorithm and the transport methods.

### Before you begin

To complete configuration of the signature verification information to secure SOAP messages, you must perform the following algorithm tasks:

- Use the WSSVerification API to configure the canonicalization and signature methods.
- Use the WSSVerifyPart API to configure the digest and transform methods.

to configure the algorithm methods to use when configuring the client for request signing.

### About this task

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 of method | Purpose |
| --- | --- |
| Digest 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 client request receiver configuration. |
| Transform algorithm | Applies to the `<Transform>` element. |
| Signature algorithm | Specifies the Uniform Resource Identifiers (URI) of the signature verification method. |
| Canonicalization algorithm | Specifies the Uniform Resource Identifiers (URI) of the canonicalization method. |

After configuring the client to digitally sign the message, you must configure the client to verify the digital signature. You can use the WSS APIs or configure policy sets using the administrative console to verify the digital signature and to choose the verification and verify part algorithms. If using the WSS APIs to configure, use the WSSVerification and WSSVerifyPart APIs to specify which digitally signed message parts to verify and to specify which algorithm methods to use when configuring the client for request signing.

The WSSVerification and WSSVerifyPart APIs perform the following steps to configure the signature verification and verify parts algorithm methods:

1. For the consumer binding, the WSSVerification API specifies the signature methods to allow for the signature verification. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSVerification.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSVerification.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
2. For the consumer binding, the WSSVerification API specifies the canonicalization method to allow for the signature verification. WebSphere Application Server supports the following pre-configured canonicalization methods by default:
   - WSSVerification.EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerification.C14N: http://www.w3.org/2001/10/xml-c14n#
3. For the consumer binding, the WSSVerifyPart API specifies the digest method, as needed. WebSphere Application Server supports the following digest method algorithms for signed parts verification:
   - WSSVerifyPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the consumer binding, the WSSVerifyPart API specifies the transform method. WebSphere Application Server supports the following transform algorithms for verify parts:
   - WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support these algorithms:
   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/TR/1999/REC-xpath-19991116

## Results

You have specified which method to use when verifying a digital signature when the client sends a message.

## Example

The following example provides sample WSS API code that specifies the verification information, the body as a part to be verified, the HMAC_SHA1 as a signature method, C14N and EXC_C14N as the candidates of canonicalization methods, TRANSFORM_STRT10 as a transform method, and SHA256 as a digest method.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance
   WSSConsumingContext concont = factory.newWSSConsumingContext();

// Generate the certificate list
```

```
   String certpath = "intca2.cer";
// The location of the X509 certificate file
      X509Certificate x509cert = null;
      try {
            InputStream is = new FileInputStream(certpath);
            CertificateFactory cf = CertificateFactory.getInstance("X.509");
           x509cert = (X509Certificate)cf.generateCertificate(is);
      } catch(FileNotFoundException e1){
          throw new WSSException(e1);
      } catch (CertificateException e2) {
          throw new WSSException(e2);
      }

      Set<Object> eeCerts = new HashSet<Object>();
      eeCerts.add(x509cert);
// Create the certStore
   java.util.List<CertStore> certList = new
        java.util.ArrayList<CertStore>();
   CollectionCertStoreParameters certparam = new
        CollectionCertStoreParameters(eeCerts);
   CertStore cert = null;
      try {
            cert = CertStore.getInstance("Collection",
                                         certparam,
                                        "IBMCertPath");
      } catch (NoSuchProviderException e1) {
           throw new WSSException(e1);
      } catch (InvalidAlgorithmParameterException e2) {
           throw new WSSException(e2);
      } catch (NoSuchAlgorithmException e3) {
           throw new WSSException (e3);
      }
      if(certList != null ){
            certList.add(cert);
      }

// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
                                "dsig-receiver.ks",
                                "jks",
                                "server".toCharArray(),
                                certList,
                                java.security.Security.getProvider(
                                      "IBMCertPath")
                                );

// Generate the WSSVerification instance
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                  callbackHandler);

// Set one or more candidates of the signature method used for
//    verification (step. 1)
// DEFAULT : WSSVerification.RSA_SHA1
      ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

// Set one or more candidates of the canonicalization method used for
//    verification (step. 2)
// DEFAULT : WSSVerification.EXC_C14N
      ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
      ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Set the part to be specified by WSSVerifyPart
   WSSVerifyPart verPart = factory.newWSSVerifyPart();

// Set the part to be specified by the keyword
      verPart.setRequiredVerifyPart(WSSVerification.BODY);

// Set the candidates of digest methods to use for verification (step. 3)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N : String
      verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);
```

```
// Set the candidates of digest methods to use for verification (step. 4)
// DEFAULT : WSSVerifyPart.SHA1
      verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

// Set WSSVerifyPart to WSSVerification
      ver.addRequiredVerifyPart(verPart);

// Add the WSSVerification to the WSSConsumingContext
   concont.add(ver);

// Validate the WS-Security header
concont.process(msgcontext);
```

### What to do next

You have completed configuring the signature verification algorithms. Next, configure the encryption or decryption algorithms, if not already configured. Or, configure the security token information, as needed.

## Signature verification methods using the WSSVerification API

You can verify the signing or signature information using the WSS API for the consumer binding. The signature and canonicalization algorithm methods are used for the generator binding. The WSSVerification API is provided in the com.ibm.websphere.wssecurity.wssapi.verification package.

To configure consumer signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

### Methods

Methods that are used for the signature verification include the:

**Signature method**
: Sets the signature algorithm method.

**Canonicalization method**
: Sets the canonicalization algorithm method.

The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the response consumer configuration.

### Signature algorithms

The signature algorithms specify the signature verification algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature verification method. WebSphere Application Server supports the following pre-configured algorithms:

*Table 44. Signature verification algorithms*

| Algorithm | Description |
| --- | --- |
| WSSVerification.HMAC_SHA1 | A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1 |
| WSSVerification.RSA_SHA1 (the default value) | A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1 |

WebSphere Application Server does not support the algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1

### Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

*Table 45. Verification canonicalization algorithms*

| Algorithm | Description |
|---|---|
| WSSVerification.C14N | A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n# |
| WSSVerification.EXC_C14N (the default value) | A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n# |

The following example provides sample WSS API code that specifies the X.509 token security token for signature verification:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();

// X509ConsumeCallbackHandler
X509ConsumeCallbackHandler callbackHandler = new
   X509ConsumeCallbackHandler("dsig-receiver.ks",
                              "jks",
                              "server".toCharArray(),
                              certList,
   java.security.Security.getProvider("IBMCertPath")46 );

// Set the verification component
// DEFAULT verification parts: Body, WS-Addressing header, and Timestamp
// DEFAULT data encryption algorithm: RSA-SHA1
// DEFAULT digest algorithm: SHA1
// DEFAULT canonicalization algorithm: exc-c14n
WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                 callbackhandler);
concont.add(ver);

// Validate the WS-Security header
concont.validate(msgctx);
```

## Choosing the verify parts methods using the WSSVerifyPart API

You can configure the signing verification information for the consumer binding using the WSS API. The transform algorithm and digest methods are used for the consumer binding. Use the WSSVerifyPart API to configure the algorithm methods. The WSSVerifyPart API is provided in the com.ibm.websphere.wssecurity.wssapi.verification package.

To configure consumer verify parts information to protect message integrity, you must first digitally sign and then verify the signature and signed parts for the SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

### Methods

Methods that are used for the signing information include the:

**Digest method**
: Sets the digest method.

**Transform method**
: Sets the transform algorithm method.

## Digest algorithms

The digest method algorithm is specified within the element is used in the <Digest> element. WebSphere Application Server supports the following pre-configured digest algorithms:

*Table 46. Verify parts digest methods*

| Digest method | Description |
|---|---|
| WSSVerifyPart.SHA1 (the default value) | A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1 |
| WSSVerifyPart.SHA256 | A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256 |
| WSSVerifyPart.SHA512 | A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512 |

## Transform algorithms

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part. WebSphere Application Server supports the following pre-configured transform algorithms:

*Table 47. Verify parts transform methods*

| Digest method | Description |
|---|---|
| WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE | A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature |
| WSSVerifyPart.TRANSFORM_STRT10 | A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform |
| WSSVerifyPart.TRANSFORM_EXC_C14N (the default value) | A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n# |
| WSSVerifyPart.TRANSFORM_XPATH2_FILTER | A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2 |

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code that verifies the body using SHA256 as the digest method and TRANSFORM_EXC_14N and TRANSFORM_STRT10 as the transform methods:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSConsumingContext instance
WSSConsumingContext concont = factory.newWSSConsumingContext();


// generate the cert list
String certpath = "intca2.cer";// The location of the X509
   certificate file X509Certificate x509cert = null;
try {
 InputStream is = new FileInputStream(certpath);
 CertificateFactory cf = CertificateFactory.getInstance("X.509");
 x509cert = (X509Certificate)cf.generateCertificate(is);
} catch(FileNotFoundException e1){
```

```
 throw new WSSException(e1);
} catch (CertificateException e2) {
 throw new WSSException(e2);
}

Set<Object> eeCerts = new HashSet<Object>();
eeCerts.add(x509cert);
// create certStore
java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
CollectionCertStoreParameters certparam = new
      CollectionCertStoreParameters(eeCerts);
CertStore cert = null;
try {
 cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
} catch (NoSuchProviderException e1) {
 throw new WSSException(e1);
} catch (InvalidAlgorithmParameterException e2) {
 throw new WSSException(e2);
} catch (NoSuchAlgorithmException e3) {
 throw new WSSException (e3);
}
if(certList != null ){
 certList.add(cert);
}

// generate callback handler
X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
  "dsig-receiver.ks",
  "jks",
  "server".toCharArray(),
  certList,
  java.security.Security.getProvider("IBMCertPath")
);

//generate WSSVerification instance
WSSVerification ver = factory.newWSSVerification(X509Token.class,
     callbackHandler);

//set one or more candidates of the signature method used for the
 //verification (step. 1)
// DEFAULT : WSSVerification.RSA_SHA1
ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

//set one or more candidates of the canonicalization method used
 //for the verification (step. 2)
// DEFAULT : WSSVerification.EXC_C14N
ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

//set the part to be specified by WSSVerifyPart
WSSVerifyPart verPart = factory.newWSSVerifyPart();

//set the part to be specified by the keyword
verPart.setRequiredVerifyPart(WSSVerification.BODY);

//set the candidates of digest methods to use for verification (step. 3)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N
verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_EXC_C14N);
verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

//set the candidates of digest methods to use for verification (step. 4)
// DEFAULT : WSSVerifyPart.SHA1
verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

//set WSSVerifyPart to WSSVerification
ver.addRequiredVerifyPart(verPart);

//add the WSSVerification to the WSSConsumingContext
```

```
concont.add(ver);

//validate the WS-Security header
concont.process(msgcontext);
```

## Validating the consumer token to protect message authenticity

The token consumer information is used on the consumer side to incorporate and validate the security token. The Username token, X509 tokens, and LTPA tokens by default are used for message authenticity.

### Before you begin

The token processing and pluggable token architecture in the Web Service Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Service Security run time.

**Note:** The com.ibm.wsspi.wssecurity.token.TokenConsumingComponent interface is not used with JAX-WS Web services. If you are using JAX-RPC Web services, this interface is still valid.

Note that the key name (KeyName) element is not supported because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification. For similar reasons, a SAML token is not supported.

### About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web services security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web services security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Service Security run time.

**Note:** WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation

- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the consumer side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching generator security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the consumer-side security token, use the appropriate pre-configured token consumer interface from the WSS APIs to complete the following token configuration process steps:

1. Generate the wssFactory instance.
2. Generate the wssConsumingContext instance.

   The WSSConsumingContext interface stores the components for consuming Web Services Security (WS-Security), such as verification, decryption, the security token, and the time stamp. When the validate() method is called, all of these components are validated.
3. Create the consumer-side components, such as the WSSVerification and the WSSDecryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.
5. Specify a token consumer class name. The token consumer class name specifies the required information to validate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specify the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   WebSphere Application Server provides the following default callback handler implementations for the consumer side:

   **com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback**
   : This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   **ccom.ibm.websphere.wssecurity.callbackhandler.UNTConsumeCallbackHandler**
   : This class is a callback handler for the Username token on the consumer side. This instance is used to set into WSSConsumingContext object to validate a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   **com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler**
   : This class is a callback handler that is used to validate the X.509 certificate that is inserted in the Web services security header within the SOAP message as a binary security token on the consumer side. This instance is used to generate the WSSVerification object and

WSSDecryption objects, set the objects into WSSConsumingContext object to validate the X.509 binary security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must have been provided on the generator side.

**com.ibm.websphere.wssecurity.callbackhandler.LTPAConsumeCallbackHandler**
This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the consumer side. This instance is used to generate the WSSVerification and WSSDecryption objects to validate an LTPA token.

This callback handler is used to validate the LTPA security token inserted in the Web services security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the Web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

**com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler**
This class is a callback handler for the Kerberos v5 token on the consumer side. This instance is used to set the WSSConsumingContext object to consume the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSVerification and WSSDecryption objects to use the Kerberos session key or derived key in the SOAP message verification and decryption.

7. If a X.509 token is specified, additional token information is also specified.

| | |
|---|---|
| keyStoreRef | The reference name of the keystore that is used for the key locator. |
| keyStorePath | The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations. |
| keyStorePassword | The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server. |
| keyStoreType | The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection:<br>**JKS** Use this option if the keystore uses the Java Keystore (JKS) format.<br>**JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption.<br>**JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).<br>**PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might contain encrypt keys that use cryptographic hardware to ensure protection.<br>**PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format. |

| | |
|---|---|
| alias | The key alias name. The key alias is used by the key locator to find the key within the keystore file. |
| keyPassword | The key password that is used for recovering the key. This password is needed to access the key object within the keystore file. |
| keyName | The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`. |
| trustAnchorPath | The file path from which the trust anchor is loaded. |
| trustAnchorType | The type of trust anchor. |
| trustAnchorPassword | The password that is used to check the integrity of the trust anchor or the password used to unlock the keystore. |
| certStores | A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). The collection certificate store is used to validate the certificate path of the incoming X.509-formatted security tokens. |
| provider | The security provider. |

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

   First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain. The names of the trust anchor and the collection certificate store are created in the certificate path under your token consumer.

d. With a keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the consumer token, and represents the token type of this class:

   **ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509**
      Specifies an X.509 certificate token.

   **ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1**
      Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web services security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

**ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7**
: Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web services security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

**ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2**
: For an LTPA token, you can use `LTPA` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

**ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2**
: For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes`http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token consumer class name, the following token information can be specified:
   a. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token consumer. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the validating token type is a Username token, and it is available only for the response consumer binding.
   b. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   c. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the consumer token, and represents the token type of this class:

   **URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken**
   : Specifies a Username token.
9. If a Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

| Token Information | Description | Default Value |
|---|---|---|
| tokenValueType | Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification. | http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ |
| requireDKT | A boolean value to require a derived key for message protection. | false |
| clabel | The client label for the derived key. | WS-SecureConversation<br><br>Specify null to use the default value. |
| slabel | The service label for the derived key. | WS-SecureConversation<br><br>Specify null to use the default value. |

| Token Information | Description | Default Value |
|---|---|---|
| keylen | The length of the derived key. | 16<br><br>Specify zero to use the default value |
| supportTokenRequireSHA1 | A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token. | false<br><br>SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed. |
| decComponent | An instance of WSSDecryption . | Set decComponent and verComponent to null to initialize this first for either the decryption or verification component. Then, use the initialized component only in the callback handler constructor for the second component. |
| verComponent | An instance of WSSVerfication. | Set decComponent and verComponent to null to initialize this first for either the decryption or verification component. Then, use the initialized component only in the callback handler constructor for the second component. |

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If secure conversation is used for message protection, then the following information must be specified:

| Information | Description |
|---|---|
| EncryptionAlgorithm | This determines the key size. |
| cLabel | The client label used when creating the derived key. |
| sLabel | The server label used when creating the derived key. |

11. Set the components into the wssConsumingContext object.
12. Invoke the wssConsumingContext.process() method.

## Results

Using the WSS APIs, you have configured the token consumer.

## What to do next

You must specify a similar token generator configuration, if not already completed.

## Configuring the consumer security tokens using the WSS API

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the consumer side, use the Web Services Security APIs (WSS API). The consumer security tokens are part of the com.ibm.websphere.wssecurity.wssapi.token interface package.

### Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Service Security run time and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must have completed the following token task: configure the generator tokens, as needed.

### About this task

On the generator side, the JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token. The token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Service Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Service Security run time to the LoginModule. After the token is authenticated and a security token object is created, then the token is passed it to the Web Service Security run time.

When using the WSS API for consumer token validation, certain default behaviors occur. The simplest way to use the WSS API is to use the default JAAS login module and callback handler. The example uses the default for them so the example does not specify the JAAS login module name.

The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide defaults for the token type, the token value, and the JAAS configuration name. The default token behaviors include:

*Table 48. Default token behaviors*

| Consumer token decisions | Default behavior |
| --- | --- |
| Which token type to use | The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type.<br><br>WebSphere Application Server provides the following pre-configured consumer token types:<br>• Security context token<br>• Derived key token<br>• X509 tokens<br><br>You can also create custom token types, as needed. |
| What JAAS login configuration name to specify | The JAAS login configuration name specifies which JAAS login configuration name to use. |
| Which configuration type to use | The JAAS login module configuration type. Only the pre-configured consumer configuration types can be used for consumer token types. |

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, XML format, and cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following token types are subclasses of the generic security token class:

*Table 49. Subclasses of the SecurityToken*

| Token type | JAAS login configuration name |
|---|---|
| Security context token | system.wss.consume.sct |
| Derived key token | system.wss.consume.dkt |

The following token types are subclasses of the binary security token class:

*Table 50. Subclasses to the BinarySecurityToken*

| Token type | JAAS login configuration name |
|---|---|
| X.509 token | system.wss.consume.x509 |
| X.509 PKI Path token | system.wss.consume.pkiPath |
| X.509 PKCS7 token | system.wss.consume.pkcs7 |

**Notes:**

- For each JAAS login token consumer configuration name, there is a respective token generator configuration name. For example, for the X509Token, the respective token generator configuration name is system.wss.generate.x509.
- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

To validate the X509Token to the SOAP message on the consumer side, the <X509Token> element must be in the <wsse:Security> element.

1. To validate the securityToken package, com.ibm.websphere.wssecurity.wssapi.token, first ensure that the application server is installed.
2. *If using the default values*, configures the tokens for the Web Services Security token consumer process. , for each token type, the process is similar to the following token consumer process:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates a JAAS CallbackHandler with information that is required to validate the security token. Review the token class information for which parameters are required or optional. For example, for an X.509 token, you could configure the following:

| keyStoreRef | Indicates the reference name of the keystore that is stored in the cryptographic card. It can be specified when the card is set to the hardware. |
|---|---|
| keyStorePath | Indicates the path of the keystore file. It is not necessary to specify the keyStorePath if the keyStoreRef is set. |
| keyStorePassword | Indicates the password of the keystore file. |
| keyStoreType | Indicates the type of keystore file. |
| alias | Indicates the alias of the key. |

| | |
|---|---|
| keyPassword | Indicates the password of the key. |
| keyName | Indicates the subject name of the key. |

d. Sets the callback handler into WSSDecryption, WSSVerification, or WSSConsumingContext.
e. If the callback handler is set into the WSSDecryption or WSSVerification, adds either one into WSSConsumingContext.
f. Calls WSSConsumingContext.process().

3. *If using other than the default values*, configures the tokens for the Web Services Security token consumer process. For each token type, the process is similar to the following token consumer process:
   a. If you do not use the default JAAS login module and callback handler, you need to prepare a custom one and register the name of JAAS login configuration using the administrative console in advance.
   b. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   c. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   d. Creates a callback handler with information that is required to validate the security token. Review the token class information for which parameters are required or optional. For example, for a X.509 token, you can configure the following:

| | |
|---|---|
| keyStoreRef | Indicates the reference name of the keystore that is stored in the cryptographic card. It can be specified when the card is set to the hardware. |
| keyStorePath | Indicates the path of the keystore file. It is not necessary to specify the keyStorePath if the keyStoreRef is set. |
| keyStorePassword | Indicates the password of the keystore file. |
| keyStoreType | Indicates the type of keystore file. |
| alias | Indicates the alias of the key. |
| keyPassword | Indicates the password of the key. |
| keyName | Indicates the subject name of the key. |

   e. Sets JAAS configuration name and callback handler into WSSDecryption or WSSVerification, or WSSConsumingContext.
   f. If JAAS configuration name and callback handler are set into the WSSDecryption or WSSVerification, adds either one into WSSConsumingContext.
   g. Calls WSSConsumingContext.process().

## Results

If there is an error condition, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and the security token on the consumer side is validated (authenticated).

## Example

The following sample code provides the WSS API example code for decryption using the default JAAS login module and callback handler:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();
```

```
// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
        X509ConsumeCallbackHandler(
                                   "",
                                   "enc-sender.jceks",
                                   "jceks",
                                   "storepass".toCharArray(),
                                   "alice",
                                   "keypass".toCharArray(),
                                   "CN=Alice, O=IBM, C=US");

// Generate the WSSDecryption instance (step: d)
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                              callbackHandler);


// Add WSSDecryption to WSSConsumingContext (step: e)
   concont.add(dec);

// Validate the WS-Security header (step: f)
concont.process(msgcontext);
```

### What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar generator tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages at the response consumer using the WSS APIs or configure the tokens using the administrative console.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by verifying the signature or by decrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

## Configuring Web services security using the WSS APIs

The Web Services Security application programming interfaces (WSS API) provide support for securing SOAP message.

### Before you begin

Web Service Security supports the following programming models:

- Programming API for securing SOAP message with Web Service Security (WSS API).

  The API programming model design has been redesigned. The new design is an interface-based programming model and is based on Web Services Security Version 1.1 standards but the design also includes support for Web Services Security Version 1.0 for securing the SOAP message. The WSS API programming model implementation is a simplified version, which is based on an early draft proposal of JSR-183, which is the JSR for defining Java API binding for Web Service Security. By design, because the application code is programmed to the interface, any application code that is programmed with the open source implementation should be able to run on the WebSphere Application Server with minimal changes or no changes at all.
- Service Programming Interfaces (SPI) for a service provider

  Similarly, the Web Service Security run time token generation and token consuming SPI have been redesign so that the same security token interface and JAAS Login Module implementation can be used for both the WSS API and the SPI. The WSS SPI for the service provider extend the security token types and provide keys and deriving keys for signing, signature verification, encryption and decryption.

## About this task

These programming models extend the following functions :

- Security token types and deriving keys for signing
- Signature and verification
- Encryption and decryption

The following figure demonstrates how to use the simplified WSS APIs to secure a SOAP message by using XML digital signature and XML encryption.

The configuration model for Web services has also been redesigned from a deployment descriptor model to a policy set model. The configuration programming model is based on configuring policy sets using a security policy to specify security constraints.

The functions provided by the policy set configurations are the same as the functions supported by the WSS API for the Web Service Security run time. However, the security policy that is defined using policy sets has a higher priority over the WSS API. When the WSS API and the policy set are both used in the application, the default behavior is for the security policy from the policy set to be enforced and the WSS API to be ignored. To use the WSS API in the application, you must make sure that there is no policy set attached to the application or to the application resources, or make sure there is no security policy in the attached policy set.

Web Service Security can be enabled by either using a policy set that is configured by using the administrative console, or by using the WSS API for configuration.

Using the WSS API, complete the following high-level steps to secure the SOAP message:

1. Use the WSSSignature API to configure the signing information for the request generator (client side) binding. Different message parts can be specified in the message protection for a request on the generator side. The default required parts are BODY, ADDRESSING_HEADERS, and TIMESTAMP.

   The WSSSignature API also specifies the different algorithm methods to be used with the signature for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.
2. Use the WSSSignPart API if you want to add or change the signed parts to be used for message protection. The default signed parts are WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS, and WSSSignature.TIMESTAMP.

   The WSSSignPart API also specifies the different algorithm methods to be used if you added or changed the signed parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N. For example, use the WSSSignPart API if you want to generate the signature for the SOAP message using the SHA256 digest method instead of the default value of SHA1.
3. Use the WSSEncryption API to configure the encryption information on the request generator side. The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The default targets of encryption are BODY_CONTENT and SIGNATURE

   The WSSEncryption API also specifies the different algorithm methods to be used to protect message confidentiality. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.
4. Use the WSSEncryptPart API if you want to add or change the encrypted parts to be used for message confidentiality. For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

   No algorithm methods are required for encrypted parts.
5. Use the WSS API to attach the token on the generator side. The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible

for creating the security token on the generator side. Different standalone tokens can be sent in request or response. The default token is the X509Token. The other token that can be used for signing is the DerivedKeyToken, which is used only with Web Services Secure Conversation (WS-SecureConversation).

6. Use the WSSVerification API to verify the signature for the response consumer (client side) binding. Different message parts can be specified in the message protection for a response on the consumer side. The required targets for verification are BODY, ADDRESSING_HEADERS, and TIMESTAMP.

   The WSSVerification API also specifies the different algorithm methods to be used for verifying the signature and for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.

7. Use the WSSVerifyPart API to add or change the verify signed parts to be used for message protection. The required verify parts are WSSVerification.BODY, WSSVerification.ADDRESSING_HEADERS, and WSSVerification.TIMESTAMP.

   The WSSVerifyPart API also specifies the different algorithm methods to be used if you added or changed the verification parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N.

8. Use the WSSDecryption API to configure the decryption information for the response consumer (client side) binding. The decryption information on the consumer side is used for decrypting an incoming SOAP message. The default targets of decryption are BODY_CONTENT and SIGNATURE. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.

   No algorithm methods are required for decryption.

9. Use the WSSDecryptPart API if you want to add or change the decrypted parts to be used for message confidentiality. For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

   No algorithm methods are required for decrypted parts.

10. Use the WSS API to configure the token on the consumer side. The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for validating (authenticating) the security token on the consumer side. Different standalone tokens can be sent in request or response.

    The WSS API adds the information for the candidate token that is used for decryption. The default token is X509Token.

### Results

### What to do next

The Web Service Security run time token generation and token consuming Service Programming Interfaces (SPI) have been redesign so that the same Security Token interface and JAAS Login Module implementation can be used in both the WSS API and the SPI. See the SPI information for detail descriptions.

## Web services security APIs

The Web services security programming model provides application programming interfaces (WSS API) for securing the SOAP message. The WSS API model is based on Web Services Security Version 1.1 standards but also includes support for Web Services Security Version 1.0.

The Web services security APIs (WSS APIs) can generate and process the following SOAP-related bindings for XML security:

- XML signature and signature verification
- XML encryption and decryption

The token processing and pluggable token architecture in the Web service security run time has been redesign to reuse the same Security Token interface and the JAAS Login Module as those used for the WSS APIs.

The following table lists the WSS API interfaces that are provided with WebSphere Application Server and used to configure signing and encryption information in the SOAP bindings for the generator and consumer bindings.

*Table 51. WSS API interfaces*

| WSS API interfaces | Description |
| --- | --- |
| WSSDecryption | Package: com.ibm.websphere.wssecurity.wssapi.decryption<br><br>This interface is responsible for specifying decryption. The default values for decryption include:<br>• Targets: BODY_CONTENT, SIGNATURE<br>• Data encryption method: AES128<br>• Key encryption method: KW_RSA_OAEP<br>• Security token: X.509 |
| WSSDecryptPart | Package: com.ibm.websphere.wssecurity.wssapi.decryption<br><br>This interface is responsible for adding decrypted parts, as needed. If specified, the default values for decrypted parts include:<br>• Security token: X.509<br>• Transform method: N/A (not applicable) |
| WSSEncryption | Package: com.ibm.websphere.wssecurity.wssapi.encryption<br><br>This interface is responsible for the encryption component. The default values for encryption include:<br>• Targets: BODY_CONTENT, SIGNATURE<br>• Data encryption method: AES128<br>• Key encryption method: KW_RSA_OAEP<br>• Security token: X.509<br>• refType: SecurityToken.REF_KEYID<br>• mtomOptimize: false |
| WSSEncryptPart | Package: com.ibm.websphere.wssecurity.wssapi.encryption<br><br>This interface is responsible for adding encrypted parts, as needed. If specified, the default values for encrypted parts include:<br>• Transform method: N/A (not applicable) |
| WSSSignature | Package: com.ibm.websphere.wssecurity.wssapi.signature<br><br>This interface is responsible for specifying the signature. The default values for signature include:<br>• Targets: BODY, ADDRESSING_HEADERS, TIMESTAMP<br>• Signature method: RSA_SHA1<br>• Canonicalization method: EXC_C14N<br>• Security token: X.509<br>• Type of token reference: SecurityToken.REF_STR |

*Table 51. WSS API interfaces (continued)*

| WSS API interfaces | Description |
| --- | --- |
| WSSSignPart | Package: com.ibm.websphere.wssecurity.wssapi.signature<br><br>This interface is responsible for adding signed parts, as needed. If specified, the default values for signed parts include:<br>• Transform method : TRANSFORM_EXC_C14N<br>• Digest method: SHA1 |
| WSSVerification | Package: com.ibm.websphere.wssecurity.wssapi.verification<br><br>This interface is responsible for specifying the signature verification. The default values for verification include:<br>• Targets: BODY, ADDRESSING_HEADERS, TIMESTAMP<br>• Signature method: RSA_SHA1<br>• Canonicalization method: EXC_C14N<br>• Security token: X.509 |
| WSSVerifyPart | Package: com.ibm.websphere.wssecurity.wssapi.verification<br><br>This interface is responsible for adding verify parts, as needed. If specified, the default values for verify parts include:<br>• Digest method: SHA1<br>• Transform method: TRANSFORM_EXC_C14N |

Also see the information about pre-configured generator and consumer tokens.

## Web services security configuration considerations when using the WSS API

To secure Web services security for WebSphere Application Server, you can specify several different configurations using the Web Services Security APIs (WSS API). The Web services security specification provides a flexible way to secure Web services messages using XML digital signature, XML encryption, and attaching security tokens. You can enable Web services security by either configuring a policy set or by using the Web services security APIs (WSS API). The implementation for WSS API has default values for which message parts are to be signed or encrypted. The default values for the WSS APIs help end users to enable Web services security quickly.

Different message parts can be specified in the message protection for request or response, and different standalone tokens can be sent in request or response. However, there is only one symmetric or one asymmetric binding assertion to describe the token type and the algorithm that is used for message protection.

Using the WSS API, you can override any default values. However, when you alter the protection parts, note that all the default protection parts are cleared. For example, if you specify that you want to encrypt the Username token instead of the default X.509 token, all the default values of the encrypting protection parts are cleared.

The following table shows an example of the relationships between each of the configurations:

*Table 52. Request generator and response consumer configurations*

| Type of configuration | Configuration name | Configurations and default values |
| --- | --- | --- |
| Request generator | Signing information | • Canonicalization method: WSSSignature.EXC_C14N<br>• Signature method: WSSSignature.RSA_SHA1<br>• Digest method: WSSSignPart.SHA1<br>• Transform method: WSSSignPart.TRANSFORM_EXC_C14N<br>• Signed part - Body: WSSSignature.BODY<br>• Signed part - Addressing: WSSSignature.ADDRESSING_HEADERS<br>• Signed part - Timestamp: WSSSignature.TIMESTAMP<br>• Token reference: SecurityToken.REF_STR<br>• Token - Value type: X509Token.ValueType<br>• Token - JAAS login configuration name: system.wss.generate.x509 |
| Response consumer | Signature verification information | • Canonicalization method: WSSVerification.EXC_C14N<br>• Signature method: WSSVerification.RSA_SHA1<br>• Transform method: WSSVerifyPart.TRANSFORM_EXC_C14N<br>• Signed part - Body: WSSVerification.BODY<br>• Signed part - Addressing: WSSVerification.ADDRESSING_HEADERS<br>• Signed part - Timestamp: WSSVerification.TIMESTAMP<br>• Token - Value type: X509Token.ValueType<br>• Token - JAAS login configuration name: system.wss.consume.x509 |
| Request generator | Encryption information | • Encrypted key: true<br>• Key encryption method: WSSEncryption.KW_RSA_OAEP<br>• Data encryption method: WSSEncryption.AES128<br>• Encryption part: WSSEncryption.BODY_CONTENT<br>• Token reference: SecurityToken.REF_KEYID<br>• Token - Value type: X509Token.ValueType<br>• Token - JAAS login configuration name: system.wss.generate.x509 |
| Response consumer | Decryption information | • Encrypted key: true<br>• Key decryption method: WSSDecryption.KW_RSA_OAEP<br>• Data decryption method: WSSDecryption.AES128<br>• Decryption part: WSSDecryption.BODY_CONTENT<br>• Token - Value type: 509Token.ValueType<br>• Token - JAAS login configuration name: system.wss.consume.x509 |

# Encrypted SOAP headers

The encrypted header element provides a standard way of encrypting SOAP headers. As one of the extensions to the OASIS SOAP message security specification, the encrypted header element indicates that the responder has processed the request. Encrypting SOAP headers and parts help to provide more secure message-level security.

The EncryptedHeader or <wsse11:EncryptedHeader> element is a part of the updated Web Services Security Version 1.1 standard and enables interoperability with other vendors that support the Version 1.1 standards, such as Microsoft .NET.

Use the EncryptedHeader element for encrypting SOAP header blocks. The EncryptedHeader element allows Web Services Security to be compliant with the SOAP mustUnderstand processing guidelines and to prevent disclosure of information that is contained in attributes on a SOAP header block.

The <wsse11:EncryptedHeader> element must contain one <xenc:EncryptedData> element. Only one <xenc:EncryptedData> element per encrypted header element is permitted.

## Encrypted data element

Normally, the programming model, such as JAX-WS, deserializes the SOAP message to a Java binding object before dispatching the call to the application code. However, if the SOAP message is encrypted, the deserialization fails because, before encryption, the original content is replaced with the EncryptedData XML element from the XML Encryption standard.

In certain cases, it might be desirable for the token that is included in the <wsse:Security> header to be encrypted for the recipient processing role.

Follow these guidelines when using the EncryptedData element:
- The EncryptedHeader element must contain one EncryptedData element.
- The <xenc:EncryptedData> element may be used to contain a security token and include it in the <wsse:Security> header.
- The <xenc:EncryptedData> must not include an XML ID for referencing the contained security token.
- All <xenc:EncryptedData> tokens must either have an embedded encryption key or must be referenced by a separate encryption key.
- If compliance with Basic Security Profile 1.1 is desired, the <xenc:EncryptedData> element must have an Id attribute.

## Policy assertion for encrypted parts

The EncryptedParts policy assertion specifies which header is to be encrypted in the security policy. The following table describes the elements and attributes that can be used for EncryptedParts.

*Table 53. Attributes and elements of the EncryptedParts element*

| Element or attribute | Description |
| --- | --- |
| /sp:EncryptedParts/sp:Header | Optional. Presence of this optional element indicates that a specific SOAP header (or set of such headers) must be protected. You can have multiple sp:Header elements within a single EncryptedParts element.<br><br>Each header (or set of headers) must be encrypted, and this encryption will encrypt the elements by using Web Services Security Version 1.1 encrypted headers. As such, if WS-Secuirty 1.1 Encrypted Headers are not supported by a service, then the headers cannot be encrypted by using message-level security.<br><br>If multiple SOAP headers with the same local name but different namespace names are to be encrypted, multiple sp:Header elements are required, either as part of a single sp:EncryptedParts assertion or as part of separate sp:EncryptedParts assertions. |
| /sp:EncryptedParts/sp:Header/@Name | Optional. This attribute indicates the local name of the SOAP header to be confidentiality protected. If this attribute is not specified, all SOAP headers whose namespace matches the Namespace attribute are to be protected. |
| /sp:EncryptedParts/sp:Header/ @Namespace | Required. This attribute indicates the namespace of the SOAP headers to be confidentiality protected. |

The following message example shows what the EncryptedHeader element looks like on a message where the EncryptedParts policy assertion for the encrypted header has been specified on the policy:

```
<S:Envelope xmlns:S="..." xmlns:wsse="..." xmlns:wsse11="..." xmlns:wsu="..."
   xmlns:xenc="..." xmlns:ds="...">
   <S:Header>
     <wsse:Security>
      <!-- Tokens etc. -->
       <xenc:EncryptedKey>
         <xenc:EncryptionMethod Algorithm="..."/>
           <ds:KeyInfo>
             ... ...
           </ds:KeyInfo>
          <xenc:CipherData>
            <xenc:CipherValue>...</xenc:CipherValue>
          </xenc:CipherData>
          <xenc:ReferenceList>
            <xenc:DataReference URI="#hdrID"/>
          </xenc:ReferenceList>
        </xenc:EncryptedKey>
     </wsse:Security>
     <wsse11:EncryptedHeader wsu:Id="hdrID">
       <xenc:EncryptedData Id="encDataID">
         <xenc:CipherData>
           <xenc:CipherValue>...</xenc:CipherValue>
         </xenc:CipherData>
         ...
       </xenc:EncryptedData>
     </wsse11:EncryptedHeader>
   </S:Header>
   <S:Body>
     ...
   </S:Body>
</S:Envelope>
```

To encrypt headers in the Web Services Security Version 1.0 specification format, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.0 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the target header for encryption is replaced by an <EncryptedData> element, instead of an <EncryptedHeader> element that contains an <EncryptedData> element.

For Web Services Security Version 1.1 behavior that is equivalent to WebSphere Application Server versions prior to version 7.0, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.1.pre.V7 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the <EncryptedHeader> element includes a wsu:Id parameter and the <EncryptedData> element omits the Id parameter. This property should only be used if compliance with Basic Security Profile 1.1 is not required.

For complete information about the EncryptedHeader element and the EncryptedData element, see the Web Services Security Version 1.1 specification.

## Signature confirmation

Web services security signature confirmation is an enhanced XML digital signature, and it is included in the Web services security standard. XML digital signature is used for signing elements of the SOAP envelope.

As one of the extensions to the OASIS SOAP message security specification, the signature confirmation element incorporates the elements that are needed within the response message in order to confirm the signature that is contained in a request message. XML digital signature and signature confirmation help to provide more secure message-level security.

Web Services Security Version 1.0 for SOAP message security did not provide any guidance on how to confirm mutual understanding of the request that prompted this response. The SignatureConfirmation or <wsse11:SignatureConfirmation> element has been added to the Web Services Security Version 1.1 specification. The <wsse11:SignatureConfirmation> element ensures that the signature is processed by the intended recipient and indicates that the responder has processed the signature in the request. The signature confirmation element is part of the updated Web Services Security standard and enables interoperability with other vendors that support the Version 1.1 standards, such as Microsoft .NET.

Because of the stateless nature of Web services and due to different message exchange patterns (MEPs), consider the following assumptions:

- Assume that session affinity is enabled if a cluster is enabled for the clients that are running in WebSphere Application Server. When session affinity is enabled, it implies that the response is sent back to the initiating client of the server.
- Assume WS-Addressing is enabled for asynchronous message exchange patterns. When WS-Addressing is enabled, it allows the run time to relate the response back to the request. An asynchronous response is sent back to the application of the initiating WebSphere Application Server.

## Syntax

The SignatureConfirmation element indicates that the responder has processed the signature in the request. When this element is not present in a response, the initiator interprets that the responder is not compliant.

The format for the signature confirmation element is as follows:

```
<wsse11:SignatureConfirmation wsu:Id="..." Value="..." />
```

where:

**wsu:Id**
The identifier that is used when referencing this element in the <ds:SignedInfo> reference list of the signature of the associated response message. This attribute is required so that unambiguous references are made to this <wsse11:SignatureConfirmation> element.

**Value**
This attribute is optional and contains the contents of a <ds:SignatureValue> that is copied from the associated request. If the request is unsigned, this attribute must not be present. If this attribute is specified without a value (empty), the initiator interprets this as incorrect behavior and processes it accordingly. When this attribute is not present, the initiator interprets this to mean that the response is based on a request that was not signed.

## Configuration

To configure signature confirmation, configure the policy file using the administrative console, and select **Require signature confirmation**. To process Signature Confirmation correctly, the initiator of the request needs to preserve the signatures during request generator processing and later needs to retrieve the signatures for confirmation checks.

## Response generation rules

Additional SOAP security elements for the SOAP responder are used to confirm that the response is in relationship to a particular request. The responder must include the contents of the <ds:SignatureValue> element of the request signature as the value of the @Value attribute of the <wsse11:SignatureConfirmation> element.

The following response generation rules apply when using the SignatureConfirmation policy assertion:

- If there are no signatures on the request, the response contains one SignatureConfirmation element, without a value. For MEPs where there are multiple requests (all without signatures) and one response, the response contains one SignatureConfirmation element without a value.
- If there are signatures on the request, the response contains a SignatureConfirmation element for each signature, with a value that matches the signature value on the request. For MEPs where there are multiple requests, with at least one containing a signature, and one response, the response contains a SignatureConfirmation element for each signature that is found on the requests, with a value that matches the signature value on the request.
- For MEPs where there is one request and multiple responses, each response contains the appropriate SignatureConfirmation elements as noted in the first and second bullets.
- If the SOAP request contains multiple signatures, the requester will find all of the signature confirmation elements contained in the response, and will check the values of the value fields of the signature confirmation elements against the values of the signatures in the original SOAP request.

# Appendix. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories infer specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

## Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

*app_client_root*
: The default installation root directory for the Java EE WebSphere Application Client is the /QIBM/ProdData/WebSphere/AppClient/V7/client directory.

*app_client_user_data_root*
: The default Java EE WebSphere Application Client user data root is the /QIBM/UserData/WebSphere/AppClient/V7/client directory.

*app_client_profile_root*
: The default Java EE WebSphere Application Client profile root is the /QIBM/UserData/WebSphere/AppClient/V7/client/profiles/*profile_name* directory.

*app_server_root*
: The default installation root directory for WebSphere Application Server is the /QIBM/ProdData/WebSphere/AppServer/V7/Base directory.

*cip_app_server_root*
: The default installation root directory is the /QIBM/ProdData/WebSphere/AppServer/V7/Base/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

  A CIP is a WebSphere Application Server product bundled with optional maintenance packages, an optional configuration archive, one or more optional enterprise archive files, and other optional files and scripts.

*cip_profile_root*
: The default profile root directory is the /QIBM/UserData/WebSphere/AppServer/V7/Base/cip/*cip_uid*/profiles/*profile_name* directory for a customized installation package (CIP) produced by the Installation Factory.

*cip_user_data_root*
: The default user data root directory is the /QIBM/UserData/WebSphere/AppServer/V7/Base/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

*if_root*
: This directory represents the root directory of the IBM WebSphere Installation Factory. Because you can download and unpack the Installation Factory to any directory on the file system to which you have write access, this directory's location varies by user. The Installation Factory is an Eclipse-based tool which creates installation packages for installing WebSphere Application Server in a reliable and repeatable way, tailored to your specific needs.

*iip_root*
: This directory represents the root directory of an *integrated installation package* (IIP) produced by the IBM WebSphere Installation Factory. Because you can create and save an IIP to any directory on the file system to which you have write access, this directory's location varies by user. An IIP is an aggregated installation package created with the Installation Factory that can include one or more generally available installation packages, one or more customized installation packages (CIPs), and other user-specified files and directories.

*java_home*
: The following directories are the root directories for all supported Java Virtual Machines (JVMs).

© IBM Corporation 2005, 2007 

| JVM | Directory |
|---|---|
| Classic JVM | /QIBM/ProdData/Java400/jdk6 |
| 32–bit IBM Technology for Java | /QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit |
| 64–bit IBM Technology for Java | /QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit |

*plugins_profile_root*
The default Web server plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver/profiles/*profile_name* directory.

*plugins_root*
The default installation root directory for Web server plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V7/webserver directory.

*plugins_user_data_root*
The default Web server plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver directory.

*product_library*
*product_lib*
This is the product library for the installed product. The product library for each Version 7.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS7*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 7.0 product installed on the system is `QWAS7A`. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, `was.install.library`, and is located under the *app_server_root* directory.

*profile_root*
The default directory for a profile named *profile_name* for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V7/Base/profiles/*profile_name* directory.

*shared_product_library*
The shared product library, which contains all of the objects shared by all installations on the system, is `QWAS7`. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

*updi_root*
The default installation root directory for the Update Installer for WebSphere Software is the /QIBM/ProdData/WebSphere/UpdateInstaller/V7/updi directory.

*user_data_root*
The default user data directory for WebSphere Application Server is the /QIBM/UserData/WebSphere/AppServer/V7/Base directory.

The profiles and profileRegistry subdirectories are created under this directory when you install the product.

*web_server_root*
The default web server path is /www/*web_server_name*.

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

© Copyright IBM Corp. 2008

# Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

© Copyright IBM Corp. 2008