BEA WebLogic Server®

Developing Applications with WebLogic Server

Version 9.0
Revised: October 6, 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

Overview of WebLogic Server Application Development

Using Ant Tasks to Configure and Use a WebLogic Server Domain

Creating a Split Development Directory Environment

Building Applications in a Split Development Directory

Deploying and Packaging from a Split Development Directory

Understanding WebLogic Server Application Classloading

Developing Applications for Production Redeployment

Creating Shared J2EE Libraries and Optional Packages

Programming Application Lifecycle Events

Programming Context Propagation

Programming JavaMail with WebLogic Server

Threading and Clustering Topics

Enterprise Application Deployment Descriptor Elements

wldeploy Ant Task Reference

Spring Applications Reference

CHAPTER 1

Overview of WebLogic Server Application Development

The following sections provide an overview of WebLogic Server® applications and basic concepts.

Document Scope and Audience

This document is written for application developers who want to build WebLogic Server e-commerce applications using the Java 2 Platform, Enterprise Edition (J2EE) from Sun Microsystems. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

WebLogic Server applications are created by Java programmers, Web designers, and application assemblers. Programmers and designers create modules that implement the business and presentation logic for the application. Application assemblers assemble the modules into applications that are ready to deploy on WebLogic Server.

WebLogic Server and the J2EE Platform

WebLogic Server implements Java 2 Platform, Enterprise Edition (J2EE) version 1.4 technologies. J2EE is the standard platform for developing multi-tier Enterprise applications based on the Java programming language. The technologies that make up J2EE were developed collaboratively by Sun Microsystems and other software vendors, including BEA Systems.

WebLogic Server J2EE applications are based on standardized, modular components. WebLogic Server provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming.

J2EE defines module behaviors and packaging in a generic, portable way, postponing run-time configuration until the module is actually deployed on an application server.

J2EE includes deployment specifications for Web applications, EJB modules, Web Services, Enterprise applications, client applications, and connectors. J2EE does not specify *how* an application is deployed on the target server—only how a standard module or application is packaged.

For each module type, the specifications define the files required and their location in the directory structure.

Note: Because J2EE is backward compatible, you can still run J2EE 1.4 applications on WebLogic Server versions 8.1 and later.

Java is platform independent, so you can edit and compile code on any platform, and test your applications on development WebLogic Servers running on other platforms. For example, it is common to develop WebLogic Server applications on a PC running Windows or Linux, regardless of the platform where the application is ultimately deployed.

For more information, refer to the J2EE 1.4 specification at: http://java.sun.com/j2ee/download.html#platformspec.

Overview of J2EE Applications and Modules

A BEA WebLogic Server™ J2EE application consists of one of the following modules or applications running on WebLogic Server:

- Web application modules—HTML pages, servlets, JavaServer Pages, and related files. See "Web Application Modules" on page 1-3.
- Enterprise Java Beans (EJB) modules—entity beans, session beans, and message-driven beans. See "Enterprise JavaBean Modules" on page 1-4.
- Connector modules—resource adapters. See "Connector Modules" on page 1-5.
- Enterprise applications—Web application modules, EJB modules, and resource adapters packaged into an application. See "Enterprise Applications" on page 1-6.

Web Application Modules

A Web application on WebLogic Server includes the following files:

- At least one servlet or JSP, along with any helper classes.
- A `web.xml` deployment descriptor, a J2EE standard XML document that describes the contents of a WAR file.
- Optionally, a `weblogic.xml` deployment descriptor, an XML document containing WebLogic Server-specific elements for Web applications.
- A Web application can also include HTML and XML pages with supporting files such as images and multimedia files.

Servlets

Servlets are Java classes that execute in WebLogic Server, accept a request from a client, process it, and optionally return a response to the client. An HttpServlet is most often used to generate dynamic Web pages in response to Web browser requests.

JavaServer Pages

JavaServer Pages (JSPs) are Web pages coded with an extended HTML that makes it possible to embed Java code in a Web page. JSPs can call custom Java classes, known as tag libraries, using HTML-like tags. The `appc` compiler compiles JSPs and translates them into servlets. WebLogic Server automatically compiles JSPs if the servlet class file is not present or is older than the JSP source file. See "Building Modules and Applications Using wlappc" on page 4-4.

You can also precompile JSPs and package the servlet class in a Web Application (WAR) file to avoid compiling in the server. Servlets and JSPs may require additional helper classes that must also be deployed with the Web application.

More Information on Web Application Modules

See:

- "Organizing J2EE Components in a Split Development Directory" on page 3-6.
- Developing Web Applications, Servlets, and JSPs for WebLogic Server
- Programming JSP Tag Extensions

Enterprise JavaBean Modules

Enterprise JavaBeans (EJBs) beans are server-side Java modules that implement a business task or entity and are written according to the EJB specification. There are three types of EJBs: session beans, entity beans, and message-driven beans.

EJB Overview

Session beans execute a particular business task on behalf of a single client during a single session. Session beans can be stateful or stateless, but are not persistent; when a client finishes with a session bean, the bean goes away.

Entity beans represent business objects in a data store, usually a relational database system. Persistence—loading and saving data—can be bean-managed or container-managed. More than just an in-memory representation of a data object, entity beans have methods that model the behaviors of the business objects they represent. Entity beans can be accessed concurrently by multiple clients and they are persistent by definition.

The container creates an instance of the message-driven bean or it assigns one from a pool to process the message. When the message is received in the JMS Destination, the message-driven

bean assigns an instance of itself from a pool to process the message. Message-driven beans are not associated with any client. They simply handle messages as they arrive.

EJBs and WebLogic Server

J2EE cleanly separates the development and deployment roles to ensure that modules are portable between EJB servers that support the EJB specification. Deploying an EJB in WebLogic Server requires running the WebLogic Server `appc` compiler to generate classes that enforce the EJB security, transaction, and life cycle policies. See "Building Modules and Applications Using wlappc" on page 4-4.

The J2EE-specified deployment descriptor, `ejb-jar.xml`, describes the enterprise beans packaged in an EJB application. It defines the beans' types, names, and the names of their home and remote interfaces and implementation classes. The `ejb-jar.xml` deployment descriptor defines security roles for the beans, and transactional behaviors for the beans' methods.

Additional deployment descriptors provide WebLogic-specific deployment information. A `weblogic-cmp-rdbms-jar.xml` deployment descriptor unique to container-managed entity beans maps a bean to tables in a database. The `weblogic-ejb-jar.xml` deployment descriptor supplies additional information specific to the WebLogic Server environment, such as JNDI bind names, clustering, and cache configuration.

For more information on Enterprise JavaBeans, see *Programming WebLogic Enterprise JavaBeans.*

Connector Modules

Connectors (also known as resource adapters) contain the Java, and if necessary, the native modules required to interact with an Enterprise Information System (EIS). A resource adapter deployed to the WebLogic Server environment enables J2EE applications to access a remote EIS. WebLogic Server application developers can use HTTP servlets, JavaServer Pages (JSPs), Enterprise Java Beans (EJBs), and other APIs to develop integrated applications that use the EIS data and business logic.

To deploy a resource adapter to WebLogic Server, you must first create and configure WebLogic Server-specific deployment descriptor, `weblogic-ra.xml` file, and add this to the deployment directory. Resource adapters can be deployed to WebLogic Server as stand-alone modules or as part of an Enterprise application. See "Enterprise Applications" on page 1-6.

For more information on connectors, see *Programming WebLogic Server Resource Adapters.*

Enterprise Applications

An Enterprise application consists of one or more Web application modules, EJB modules, and resource adapters. It might also include a client application. An Enterprise application is defined by an `application.xml` file, which is the standard J2EE deployment descriptor for Enterprise applications. If the application includes WebLogic Server-specific extensions, the application is further defined by a `weblogic-application.xml` file. Enterprise Applications that include a client module will also have a `client-application.xml` deployment descriptor and a WebLogic run-time client application deployment descriptor. See Appendix A, "Enterprise Application Deployment Descriptor Elements."

For both production and development purposes, BEA recommends that you package and deploy even stand-alone Web applications, EJBs, and resource adapters as part of an Enterprise application. Doing so allows you to take advantage of BEA's new split development directory structure, which greatly facilities application development. See Chapter 3, "Creating a Split Development Directory Environment."

An Enterprise application consists of Web application modules, EJB modules, and resource adapters. It can be packaged as follows:

- For development purposes, BEA recommends the WebLogic split development directory structure. Rather than having a single archived EAR file or an exploded EAR directory structure, the split development directory has two parallel directories that separate source files and output files. This directory structure is optimized for development on a single WebLogic Server instance. See Chapter 3, "Creating a Split Development Directory Environment." BEA provides the `wlpackage` Ant task, which allows you to create an EAR without having to use the JAR utility; this is exclusively for the split development directory structure. See "Packaging Applications Using wlpackage" on page 5-2.
- For development purposes, BEA further recommends that you package stand-alone Web applications and Enterprise JavaBeans (EJBs) as part of an Enterprise application, so that you can take advantage of the split development directory structure. See "Organizing J2EE Components in a Split Development Directory" on page 3-6.
- For production purposes, BEA recommends the exploded (unarchived) directory format. This format enables you to update files without having to redeploy the application. To update an archived file, you must unarchive the file, update it, then rearchive and redeploy it.
- You can choose to package your application as a JAR archived file using the `jar` utility with an `.ear` extension. Archived files are easier to distribute and take up less space. An EAR file contains all of the JAR, WAR, and RAR module archive files for an application

and an XML descriptor that describes the bundled modules. See "Packaging Applications Using wlpackage" on page 5-2.

The `META-INF/application.xml` deployment descriptor contains an element for each Web application, EJB, and connector module, as well as additional elements to describe security roles and application resources such as databases. See Appendix A, "Enterprise Application Deployment Descriptor Elements."

WebLogic Web Services

Web services can be shared by and used as modules of distributed Web-based applications. They commonly interface with existing back-end applications, such as customer relationship management systems, order-processing systems, and so on. Web services can reside on different computers and can be implemented by vastly different technologies, but they are packaged and transported using standard Web protocols, such as HTTP, thus making them easily accessible by any user on the Web. See *Programming WebLogic Web Services.*

A Web service consists of the following modules:

- A Web Service implementation hosted by a server on the Web. WebLogic Web Services are hosted by WebLogic Server. A Web Service module may include either Java classes or EJBs that implement the Web Service. Web Services are packaged either as Web Application archives (WARs) or EJB modules (JARs) depending on the implementation. See *Programming WebLogic Web Services* for more information.
- A standard for transmitting data and Web service invocation calls between the Web service and the user of the Web service. WebLogic Web Services use Simple Object Access Protocol (SOAP) 1.1 as the message format and HTTP as the connection protocol.
- A standard for describing the Web service to clients so they can invoke it. WebLogic Web Services use Web Services Description Language (WSDL) 1.1, an XML-based specification, to describe themselves.
- A standard for clients to invoke Web services (JAX-RPC).
- A standard for finding and registering the Web service (UDDI).

XML Deployment Descriptors

Modules and applications have deployment descriptors—XML documents—that describe the contents of the directory or JAR file. Deployment descriptors are text documents formatted with XML tags. The J2EE specifications define standard, portable deployment descriptors for J2EE

modules and applications. BEA defines additional WebLogic-specific deployment descriptors for deploying a module or application in the WebLogic Server environment.

Table 1-1 lists the types of modules and applications and their J2EE-standard and WebLogic-specific deployment descriptors.

Table 1-1 J2EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Web Application	J2EE	`web.xml` See the Sun Microsystems Servlet 2.4 Schema.
	WebLogic	`weblogic.xml` Schema: http://www.bea.com/ns/weblogic/90/weblogic-web-app.xsd See "weblogic.xml Deployment Descriptor Elements" in *Developing Web Applications for WebLogic Server* for more information.
Enterprise Bean	J2EE	`ejb-jar.xml` See the Sun Microsystems EJB 2.1 Schema.

	WebLogic	`weblogic-ejb-jar.xml`
		Schema: http://www.bea.com/ns/weblogic/90/weblogic-ejb-jar.xsd
		See "The weblogic-ejb-jar.xml Deployment Descriptor" in *Programming WebLogic Enterprise JavaBeans*.
		`weblogic-cmp-rdbms-jar.xml`
		Schema: http://www.bea.com/ns/weblogic/90/weblogic-rdbms20-persistence.xsd
		See "The weblogic-cmp-rdbms-jar.xml Deployment Descriptor" in *Programming WebLogic Enterprise JavaBeans*.

Web Services	J2EE	`webservices.xml` See the Sun Microsystems Web Services 1.1 Schema.
	WebLogic	`weblogic-webservices.xml` Schema: http://www.bea.com/ns/weblogic/90/weblogic-wsee.xsd See "WebLogic Web Service Deployment Descriptor Element Reference" in *Programming Web Services for WebLogic Server*.

Table 1-1 J2EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Resource Adapter	J2EE	`ra.xml` See the Sun Microsystems Connector 1.5 Schema.
	WebLogic	`weblogic-ra.xml` Schema: http://www.bea.com/ns/weblogic/90/weblogic-ra.xsd See "weblogic-ra.xml Schema" in *Programming WebLogic Server Resource Adapters*.
Enterprise Application	J2EE	`application.xml` See the Sun Microsystems Application 1.4 Schema.
	WebLogic	`weblogic-application.xml` Schema: http://www.bea.com/ns/weblogic/90/weblogic-application.xsd See "weblogic-application.xml Deployment Descriptor Elements" on page A-1.
Client Application	J2EE	`application-client.xml` See the Sun Microsystems Application Client 1.4 Schema.
	WebLogic	`weblogic-appclient.xml` Schema: http://www.bea.com/ns/weblogic/90/weblogic-appclient.xsd See *Programming Stand-alone Clients*.

Note: The XML Schemas for the WebLogic deployment descriptors listed in the preceding table include elements from the weblogic-j2ee.xsd Schema, which describes common elements shared among all WebLogic-specific deployment descriptors.

When you package a module or application, you create a directory to hold the deployment descriptors—`WEB-INF` or `META-INF`—and then create the XML deployment descriptors in that directory.

Automatically Generating Deployment Descriptors

WebLogic Server provides a variety of tools for automatically generating deployment descriptors. These are discussed in the sections that follow.

EJBGen

EJBGen is an Enterprise JavaBeans 2.0 code generator or command-line tool that uses Javadoc markup to generate EJB deployment descriptor files. You annotate your Bean class file with Javadoc tags and then use EJBGen to generate the Remote and Home classes and the deployment descriptor files for an EJB application, reducing to a single file you need to edit and maintain your EJB `.java` and descriptor files. See "EJBGen Reference" in *Programming WebLogic Enterprise JavaBeans*.

Java-based Command-line Utilities

WebLogic Server includes a set of Java-based command-line utilities that automatically generate both standard J2EE and WebLogic-specific deployment descriptors for Web applications and Enterprise Applications.

These command-line utilities examine the classes you have assembled in a staging directory and build the appropriate deployment descriptors based on the servlet classes, and so on. These utilities include:

- `java weblogic.marathon.ddinit.EARInit`—automatically generates the deployment descriptors for Enterprise applications.
- `java weblogic.marathon.ddinit.WebInit`—automatically generates the deployment descriptors for Web applications.

For an example of `DDInit`, assume that you have created a directory called `c:\stage` that contains the JSP files and other objects that make up a Web application but you have not yet created the `web.xml` and `weblogic.xml` deployment descriptors. To automatically generate them, execute the following command:

```
prompt> java weblogic.marathon.ddInit.WebInit c:\stage
```

The utility generates the `web.xml` and `weblogic.xml` deployment descriptors and places them in the `WEB-INF` directory, which `DDInit` will create if it does not already exist.

Upgrading Deployment Descriptors From Previous Releases of J2EE and WebLogic Server

So that your applications can take advantage of the features in the current J2EE specification and release of WebLogic Server, BEA recommends that you always upgrade deployment descriptors when you migrate applications to a new release of WebLogic Server.

To upgrade the deployment descriptors in your J2EE applications and modules, first use the `weblogic.DDConverter` tool to generate the upgraded descriptors into a temporary directory. Once you have inspected the upgraded deployment descriptors to ensure that they are correct, repackage your J2EE module archive or exploded directory with the new deployment descriptor files.

Invoke `weblogic.DDConverter` with the following command:

```
java weblogic.DDConverter [options] archive_file_or_directory
```

where *`archive_file_or_directory`* refers to the archive file (EAR, WAR, JAR, or RAR) or exploded directory of your Enterprise application, Web application, EJB, or resource adapter.

The following table describes the `weblogic.DDConverter` command options.

Table 1-2 `weblogic.DDConverter` Command Options

Option	Description
`-d <dir>`	Specifies the directory to which descriptors are written.
`-help`	Prints the standard usage message.
`-quiet`	Turns off output messages except error messages.
`-verbose`	Turns on additional output used for debugging.

The following example shows how to use the `weblogic.DDConverter` command to generate upgraded deployment descriptors for the `my.ear` Enterprise application into the subdirectory `tempdir` in the current directory:

```
java weblogic.DDConverter -d tempdir my.ear
```

Development Software

This section reviews required and optional tools for developing WebLogic Server applications.

Apache Ant

The preferred BEA method for building applications with WebLogic Server is Apache Ant. Ant is a Java-based build tool. One of the benefits of Ant is that is it is extended with Java classes, rather than shell-based commands. BEA provides numerous Ant extension classes to help you compile, build, deploy, and package applications using the WebLogic Server split development directory environment.

Another benefit is that Ant is a cross-platform tool. Developers write Ant build scripts in eXtensible Markup Language (XML). XML tags define the targets to build, dependencies among targets, and tasks to execute in order to build the targets. Ant libraries are bundled with WebLogic Server to make it easier for our customers to build Java applications out of the box.

To use Ant, you must first set your environment by executing either the `setExamplesEnv.cmd` (Windows) or `setExamplesEnv.sh` (UNIX) commands located in the *`WL_SERVER`*`\samples\domains\wl_server` directory, where *`WL_SERVER`* is your WebLogic Server installation directory.

For a complete explanation of ant capabilities, see:
http://jakarta.apache.org/ant/manual/index.html

Note: The Apache Jakarta Web site publishes online documentation for only the most current version of Ant, which might be different from the version of Ant that is bundled with WebLogic Server. Use the following command, after setting your WebLogic environment, to determine the version of Ant bundled with WebLogic Server:

```
prompt> ant -version
```

To view the documentation for a specific version of Ant, such as the version included with WebLogic Server, download the Ant zip file from http://archive.apache.org/dist/ant/binaries/ and extract the documentation.

For more information on using Ant to compile your cross-platform scripts or using cross-platform scripts to create XML scripts that can be processed by Ant, refer to any of the WebLogic Server examples, such as *`WL_HOME`*`/samples/server/examples/src/examples/ejb20/basic/beanManaged/build.xml`.

Also refer to the following WebLogic Server documentation on building examples using Ant: *`WL_HOME`*`/samples/server/examples/src/examples/examples.html`.

Using A Third-Party Version of Ant

You can use your own version of Ant if the one bundled with WebLogic Server is not adequate for your purposes. To determine the version of Ant that is bundled with WebLogic Server, run the following command after setting your WebLogic environment:

```
prompt> ant -version
```

If you plan to use a different version of Ant, you can replace the appropriate JAR file in the *WL_HOME*\server\lib\ant directory with an updated version of the file (where *WL_HOME* refers to the main WebLogic installation directory, such as c:\bea\weblogic90) or add the new file to the front of your CLASSPATH.

Changing the Ant Heap Size

By default the environment script allocates a heap size of 128 megabytes to Ant. You can increase or decrease this value for your own projects by setting the -X option in your local ANT_OPTS environment variable. For example:

```
prompt> setenv ANT_OPTS=-Xmx128m
```

If you want to set the heap size permanently, add or update the MEM_ARGS variable in the scripts that set your environment, start WebLogic Server, and so on, as shown in the following snippet from a Windows command script that starts a WebLogic Server instance:

```
set MEM_ARGS=-Xms32m -Xmx200m
```

See the scripts and commands in *WL_HOME*/server/bin for examples of using the MEM_ARGS variable.

Source Code Editor or IDE

You need a text editor to edit Java source files, configuration files, HTML or XML pages, and JavaServer Pages. An editor that gracefully handles Windows and UNIX line-ending differences is preferred, but there are no other special requirements for your editor. You can edit HTML or XML pages and JavaServer Pages with a plain text editor, or use a Web page editor such as DreamWeaver. For XML pages, you can also use BEA XML Editor. See *BEA dev2dev Online* at http://dev2dev.bea.com/index.jsp.

Database System and JDBC Driver

Nearly all WebLogic Server applications require a database system. You can use any DBMS that you can access with a standard JDBC driver, but services such as WebLogic Java Message Service (JMS) require a supported JDBC driver for Oracle, Sybase, Informix, Microsoft SQL

Server, IBM DB2, or PointBase. Refer to *Platform Support* to find out about supported database systems and JDBC drivers.

Web Browser

Most J2EE applications are designed to be executed by Web browser clients. WebLogic Server supports the HTTP 1.1 specification and is tested with current versions of the Netscape Communicator and Microsoft Internet Explorer browsers.

When you write requirements for your application, note which Web browser versions you will support. In your test plans, include testing plans for each supported version. Be explicit about version numbers and browser configurations. Will your application support Secure Socket Layers (SSL) protocol? Test alternative security settings in the browser so that you can tell your users what choices you support.

If your application uses applets, it is especially important to test browser configurations you want to support because of differences in the JVMs embedded in various browsers. One solution is to require users to install the Java plug-in from Sun so that everyone has the same Java run-time version.

Third-Party Software

You can use third-party software products to enhance your WebLogic Server development environment. See *BEA WebLogic Developer Tools Resources*, which provides developer tools information for products that support the BEA application servers.

To download some of these tools, see *BEA WebLogic Server Downloads* at `http://commerce.bea.com/downloads/weblogic_server_tools.jsp`.

Note: Check with the software vendor to verify software compatibility with your platform and WebLogic Server version.

CHAPTER 2

Using Ant Tasks to Configure and Use a WebLogic Server Domain

The following sections describe how to start and stop WebLogic Server instances and configure WebLogic Server domains using WebLogic Ant tasks that you can include in your development build scripts:

Overview of Configuring and Starting Domains Using Ant Tasks

WebLogic Server provides a pair of Ant tasks to help you perform common configuration tasks in a development environment. The configuration tasks enable you to start and stop WebLogic Server instances as well as create and configure WebLogic Server domains.

When combined with other WebLogic Ant tasks, you can create powerful build scripts for demonstrating or testing your application with custom domains. For example, a single Ant build script can:

- Compile your application using the `wlcompile`, `wlappc`, and Web Services Ant tasks.
- Create a new single-server domain and start the Administration Server using the `wlserver` Ant task.
- Configure the new domain with required application resources using the `wlconfig` Ant task.
- Deploy the application using the `wldeploy` Ant task.
- Automatically start a compiled client application to demonstrate or test product features.

The sections that follow describe how to use the configuration Ant tasks, `wlserver` and `wlconfig`.

Starting Servers and Creating Domains Using the wlserver Ant Task

The `wlserver` Ant task enables you to start, reboot, shutdown, or connect to a WebLogic Server instance. The server instance may already exist in a configured WebLogic Server domain, or you can create a new single-server domain for development by using the `generateconfig=true` attribute.

When you use the `wlserver` task in an Ant script, the task does not return control until the specified server is available and listening for connections. If you start up a server instance using wlserver, the server process automatically terminates after the Ant VM terminates. If you only connect to a currently-running server using the `wlserver` task, the server process keeps running after Ant completes.

The `wlserver` WebLogic Server Ant task extends the standard `java` Ant task (`org.apache.tools.ant.taskdefs.Java`). This means that all the attributes of the `java` Ant task also apply to the `wlserver` Ant task. For example, you can use the `output` and `error` attributes to specify the name of the files to which output and standard errors of the `wlserver`

Ant task is written, respectively. For full documentation about the attributes of the standard java Ant task, see Java on the Apache Ant site.

Basic Steps for Using wlserver

To use the wlserver Ant task:

1. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory `WL_HOME\server\bin`, where `WL_HOME` is the top-level directory of your WebLogic Server installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory `WL_HOME/server/bin`, where `WL_HOME` is the top-level directory of your WebLogic Server installation.

   Note: The `wlserver` task is predefined in the version of Ant shipped with WebLogic Server. If you want to use the task with your own Ant installation, add the following task definition in your build file:

   ```
   <taskdef name="wlserver"
   classname="weblogic.ant.taskdefs.management.WLServer"/>
   ```

2. Add a call to the `wlserver` task in the build script to start, shutdown, restart, or connect to a server. See "wlserver Ant Task Reference" on page 2-4 for information about `wlserver` attributes and default behavior.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   ```
   prompt> ant
   ```

   Use `ant -verbose` to obtain more detailed messages from the `wlserver` task.

Sample build.xml Files for wlserver

The following shows a minimal `wlserver` target that starts a server in the current directory using all default values:

```
<target name="wlserver-default">
  <wlserver/>
</target>
```

This target connects to an existing, running server using the indicated connection parameters and username/password combination:

```
  <target name="connect-server">
    <wlserver host="127.0.0.1" port="7001" username="weblogic"
password="weblogic" action="connect"/>
  </target>
```

This target starts a WebLogic Server instance configured in the `config` subdirectory:

```
  <target name="start-server">
    <wlserver dir="./config" host="127.0.0.1" port="7001" action="start"/>
  </target>
```

This target creates a new single-server domain in an empty directory, and starts the domain's server instance:

```
  <target name="new-server">
    <delete dir="./tmp"/>
    <mkdir dir="./tmp"/>
    <wlserver dir="./tmp" host="127.0.0.1" port="7001"
    generateConfig="true" username="weblogic" password="weblogic"
action="start"/>
  </target>
```

wlserver Ant Task Reference

The following table describes the attributes of the `wlserver` Ant task.

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
policy	The path to the security policy file for the WebLogic Server domain. This attribute is used only for starting server instances.	File	No
dir	The path that holds the domain configuration (for example, `c:\bea\user_projects\mydomain`). By default, `wlserver` uses the current directory.	File	No
beahome	The path to the BEA home directory (for example, `c:\bea`).	File	No
weblogichome	The path to the WebLogic Server installation directory (for example, `c:\bea\weblogic81`).	File	No

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
servername	The name of the server to start, shutdown, reboot, or connect to. A WebLogic Server instance is uniquely identified by its protocol, host, and port values, so if you use this set of attributes to specify the server you want to start, shutdown or reboot, you do not need to specify its actual name using the `servername` attribute. The only exception is when you want to shutdown the Administration server; in this case you *must* specify this attribute. The default value for this attribute is `myserver`.	String	Required only when shutting down the Administration server.
domainname	The name of the WebLogic Server domain in which the server is configured.	String	No
adminserverurl	The URL to access the Administration Server in the domain. This attribute is required if you are starting up a Managed Server in the domain.	String	Required for starting Managed Servers.
username	The username of an administrator account. If you omit both the `username` and `password` attributes, `wlserver` attempts to obtain the encrypted username and password values from the `boot.properties` file. See Boot Identity Files in the *Managing Server Startup and Shutdown* for more information on `boot.properties`.	String	No
password	The password of an administrator account. If you omit both the `username` and `password` attributes, `wlserver` attempts to obtain the encrypted username and password values from the `boot.properties` file. See Boot Identity Files in the *Managing Server Startup and Shutdown* for more information on `boot.properties`.	String	No
pkpassword	The private key password for decrypting the SSL private key file.	String	No

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
timeout	The maximum time, in milliseconds, that `wlserver` waits for a server to boot. This also specifies the maximum amount of time to wait when connecting to a running server. The default value for this attribute is `0`, which means the Ant task never times out.	long	No
timeoutSeconds	The maximum time, in seconds, that `wlserver` waits for a server to boot. This also specifies the maximum amount of time to wait when connecting to a running server. The default value for this attribute is `0`, which means the Ant task never times out.	long	No
productionmodeenabled	Specifies whether a server instance boots in development mode or in production mode. Development mode enables a WebLogic Server instance to automatically deploy and update applications that are in the *`domain_name`*`/autodeploy` directory (where *`domain_name`* is the name of a WebLogic Server domain). In other words, development mode lets you use auto-deploy. Production mode disables the auto-deployment feature. See *Deploying Applications and Modules* for more information. Valid values for this attribute are `True` and `False`. The default value is `False` (which means that by default a server instance boots in development mode.) **Note:** If you boot the server in production mode by setting this attribute to `True`, you must reboot the server to set the mode back to development mode. Or in other words, you cannot reset the mode on a running server using other administrative tools, such as the WebLogic Server Scripting Tool (WLST).	boolean	No
host	The DNS name or IP address on which the server instance is listening. The default value for this attribute is `localhost`.	String	No

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
port	The TCP port number on which the server instance is listening. The default value for this attribute is `7001`.	int	No
generateconfig	Specifies whether or not `wlserver` creates a new domain for the specified server. Valid values for this attribute are `true` and `false`. The default value is `false`.	boolean	No
action	Specifies the action `wlserver` performs: `start`, `shutdown`, `reboot`, or `connect`. The `shutdown` action can be used with the optional `forceshutdown` attribute perform a forced shutdown. The default value for this attribute is `start`.	String	No
failonerror	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No
forceshutdown	This optional attribute is used in conjunction with the `action="shutdown"` attribute to perform a forced shutdown. For example: `<wlserver host="${wls.host}" port="${port}" username="${wls.username}" password="${wls.password}" action="shutdown" forceshutdown="true"/>` Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No
protocol	Specifies the protocol that the `wlserver` Ant task uses to communicate with the WebLogic Server instance. Valid values are `t3`, `t3s`, `http`, `https`, and `iiop`. The default value is `t3`.	String	No

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
forceImplicitUpgrade	Specifies whether the `wlserver` Ant task, if run against an 8.1 (or previous) domain, should implicitly upgrade it to version 9.0. Valid values are `true` or `false`. The default value is `false`, which means that the Ant task does *not* implicitly upgrade the domain, but rather, will fail with an error indicating that the domain needs to be upgraded to version 9.0 of WebLogic Server. For more information about upgrading domains, see Upgrading WebLogic Application Environments.	Boolean	No.
configFile	Specifies the configuration file for your domain. The value of this attribute must be a valid XML file that conforms to the XML schema as defined in the BEA WebLogic Server Configuration Reference. The XML file must exist in the Administration Server's root directory, which is either the current directory or the directory that you specify with the `dir` attribute. If you do not specify this attribute, the default value is `config.xml` in the directory specified by the `dir` attribute. If you do not specify the dir attribute, then the default domain directory is the current directory.	String	No.

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
useBootProperties	Specifies whether to use the `boot.properties` file when starting a WebLogic Server instance. If this attribute is set to `true`, WebLogic Server uses the username and encrypted password stored in the `boot.properties` file to start rather than any values set with the `username` and `password` attributes. **Note:** The values of the `username` and `password` attributes are still used when shutting down or rebooting the WebLogic Server instance. The `useBootProperties` attribute applies *only* when starting the server. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No
verbose	Specifies that the Ant task output additional information as it is performing its action. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No

Configuring a WebLogic Server Domain Using the wlconfig Ant Task

The following sections describe how to use the `wlconfig` Ant task to configure a WebLogic Server domain.

What the wlconfig Ant Task Does

The `wlconfig` Ant task enables you to configure a WebLogic Server domain by creating, querying, or modifying configuration MBeans on a running Administration Server instance. Specifically, `wlconfig` enables you to:

- Create new MBeans, optionally storing the new MBean Object Names in Ant properties.
- Set attribute values on a named MBean available on the Administration Server.

- Create MBeans and set their attributes in one step by nesting set attribute commands within create MBean commands.
- Query MBeans, optionally storing the query results in an Ant property reference.
- Query MBeans and set attribute values on all matching results.
- Establish a parent/child relationship among MBeans by nesting create commands within other create commands.

Warning: The `wlconfig` Ant task works *only* against MBeans that are in the compatitibility MBean server, which has been deprecated as of version 9.0 of WebLogic Server.

In particular, the `wlconfig` Ant task uses the deprecated BEA proprietary API `weblogic.management.MBeanHome` to access WebLogic MBeans, the same as it did in Version 8.1 of WebLogic Server. The Ant task does *not* use the standard JMX interface (`javax.management.MBeanServerConnection`) to discover MBeans.

This means that the only MBeans that you can access using `wlconfig` are those listed under the Deprecated MBeans category in the WebLogic Server MBean Reference.

Basic Steps for Using wlconfig

1. Set your environment in a command shell. See "Basic Steps for Using wlserver" on page 2-3 for details.

   Note: The `wlconfig` task is predefined in the version of Ant shipped with WebLogic Server. If you want to use the task with your own Ant installation, add the following task definition in your build file:

   ```
   <taskdef name="wlconfig"
   classname="weblogic.ant.taskdefs.management.WLConfig"/>
   ```

2. `wlconfig` is commonly used in combination with `wlserver` to configure a new WebLogic Server domain created in the context of an Ant task. If you will be using `wlconfig` to configure such a domain, first use `wlserver` attributes to create a new domain and start the WebLogic Server instance.

3. Add an initial call to the `wlconfig` task to connect to the Administration Server for a domain. For example:

   ```
   <target name="doconfig">
      <wlconfig url="t3://localhost:7001" username="weblogic"
   ```

```
        password="weblogic">
    </target>
```

4. Add nested `create`, `delete`, `get`, `set`, and `query` elements to configure the domain.
5. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   ```
   prompt> ant doconfig
   ```

   Use `ant -verbose` to obtain more detailed messages from the `wlconfig` task.

Sample build.xml Files for wlconfig

The following sections provide sample Ant build scripts for using the `wlconfig` Ant task.

Complete Example

This example shows a single `build.xml` file that creates a new domain using `wlserver` and performs various domain configuration tasks with `wlconfig`. The configuration tasks set up domain resources required by the Avitek Medical Records sample application.

The script starts by creating the new domain:

```
<target name="medrec.config">
   <mkdir dir="config"/>
   <wlserver username="a" password="a" servername="MedRecServer"
      domainname="medrec" dir="config" host="localhost" port="7000"
      generateconfig="true"/>
```

The script then starts the `wlconfig` task by accessing the newly-created server:

```
   <wlconfig url="t3://localhost:7000" username="a" password="a">
```

Within the `wlconfig` task, the `query` element runs a query to obtain the Server MBean object name, and stores this MBean in the `${medrecserver}` Ant property:

```
    <query domain="medrec" type="Server" name="MedRecServer"
         property="medrecserver"/>
```

The script the uses a `create` element to create a new JDBC connection pool in the domain, storing the object name in the `${medrecpool}` Ant property. Nested `set` elements in the `create` operation set attributes on the newly-created MBean. The new pool is target to the server using the `${medrecserver}` Ant property set in the query above:

```
<create type="JDBCConnectionPool" name="MedRecPool"
   property="medrecpool">
   <set attribute="CapacityIncrement" value="1"/>
   <set attribute="DriverName"
      value="com.pointbase.jdbc.jdbcUniversalDriver"/>
   <set attribute="InitialCapacity" value="1"/>
   <set attribute="MaxCapacity" value="10"/>
   <set attribute="Password" value="MedRec"/>
   <set attribute="Properties" value="user=MedRec"/>
   <set attribute="RefreshMinutes" value="0"/>
   <set attribute="ShrinkPeriodMinutes" value="15"/>
   <set attribute="ShrinkingEnabled" value="true"/>
   <set attribute="TestConnectionsOnRelease" value="false"/>
   <set attribute="TestConnectionsOnReserve" value="false"/>
   <set attribute="URL"
      value="jdbc:pointbase:server://localhost/demo"/>
   <set attribute="Targets" value="${medrecserver}"/>
</create>
```

Next, the script creates a JDBC TX DataSource using the JDBC connection pool created above:

```
<create type="JDBCTxDataSource" name="Medical Records Tx DataSource">
   <set attribute="JNDIName" value="MedRecTxDataSource"/>
   <set attribute="PoolName" value="MedRecPool"/>
   <set attribute="Targets" value="${medrecserver}"/>
</create>
```

The script creates a new JMS connection factory using nested `set` elements:

```
<create type="JMSConnectionFactory" name="Queue">
   <set attribute="JNDIName" value="jms/QueueConnectionFactory"/>
   <set attribute="XAServerEnabled" value="true"/>
   <set attribute="Targets" value="${medrecserver}"/>
</create>
```

A new JMS JDBC store is created using the `MedRecPool`:

```
<create type="JMSJDBCStore" name="MedRecJDBCStore"
   property="medrecjdbcstore">
   <set attribute="ConnectionPool" value="${medrecpool}"/>
```

```
      <set attribute="PrefixName" value="MedRec"/>
   </create>
```

When creating a new JMS server, the script uses a nested `create` element to create a JMS queue, which is the child of the JMS server:

```
   <create type="JMSServer" name="MedRecJMSServer">
      <set attribute="Store" value="${medrecjdbcstore}"/>
      <set attribute="Targets" value="${medrecserver}"/>
      <create type="JMSQueue" name="Registration Queue">
         <set attribute="JNDIName" value="jms/REGISTRATION_MDB_QUEUE"/>
      </create>
   </create>
```

This script creates a new mail session and startup class:

```
   <create type="MailSession" name="Medical Records Mail Session">
      <set attribute="JNDIName" value="mail/MedRecMailSession"/>
      <set attribute="Properties"
         value="mail.user=joe;mail.host=mail.mycompany.com"/>
      <set attribute="Targets" value="${medrecserver}"/>
   </create>

   <create type="StartupClass" name="StartBrowser">
      <set attribute="Arguments" value="port=${listenport}"/>
      <set attribute="ClassName"
         value="com.bea.medrec.startup.StartBrowser"/>
      <set attribute="FailureIsFatal" value="false"/>
      <set attribute="Notes" value="Automatically starts a browser on
server boot."/>

      <set attribute="Targets" value="${medrecserver}"/>

    </create>
```

Finally, the script obtains the `WebServer` MBean and sets the log filename using a nested `set` element:

```
      <query domain="medrec" type="WebServer" name="MedRecServer">
         <set attribute="LogFileName" value="logs/access.log"/>
      </query>
   </wlconfig>
</target>
```

Query and Delete Example

The `query` element does not need to specify an MBean name when nested within a `query` element:

```
<target name="queryDelete">
   <wlconfig url="${adminurl}" username="${user}" password="${pass}"
      failonerror="false">
      <query query="${wlsdomain}:Name=MyNewServer2,*"
         property="deleteQuery">
         <delete/>
      </query>
   </wlconfig>
</target>
```

Example of Setting Multiple Attribute Values

The `set` element allows you to set an attribute value to multiple object names stored in Ant properties. For example, the following target stores the object names of two servers in separate Ant properties, then uses those properties to assign both servers to the target attribute of a new JDBC Connection Pool:

```
<target name="multipleJDBCTargets">
   <wlconfig url="${adminurl}" username="${user}" password="${pass}">
      <query domain="mydomain" type="Server" name="MyServer"
         property="myserver"/>
      <query domain="mydomain" type="Server" name="OtherServer"
         property="otherserver"/>
      <create type="JDBCConnectionPool" name="sqlpool" property="sqlpool">
         <set attribute="CapacityIncrement" value="1"/>
[.....]
         <set attribute="Targets" value="${myserver};${otherserver}"/>
      </create>
   </wlconfig>
</target>
```

wlconfig Ant Task Reference

The following sections describe the attributes and elements that can be used with `wlconfig`.

Main Attributes

The following table describes the main attributes of the `wlconfig` Ant task.

Table 2-2 Main Attributes of the wlconfig Ant Task

Attribute	Description	Data Type	Required?
url	The URL of the domain's Administration Server.	String	Yes
username	The username of an administrator account.	String	No
password	The password of an administrator account. To avoid having the plain text password appear in the build file or in process utilities such as `ps`, first store a valid username and encrypted password in a configuration file using the `weblogic.Admin STOREUSERCONFIG` command. Then omit both the `username` and `password` attributes in your Ant build file. When the attributes are omitted, `wlconfig` attempts to login using values obtained from the default configuration file. If you want to obtain a username and password from a non-default configuration file and key file, use the `userconfigfile` and `userkeyfile` attributes with `wlconfig`. See STOREUSERCONFIG in the *WebLogic Server Command Reference* for more information on storing and encrypting passwords.	String	No
failonerror	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. This attribute is set to true by default.	Boolean	No

Table 2-2 Main Attributes of the wlconfig Ant Task

Attribute	Description	Data Type	Required?
userconfigfile	Specifies the location of a user configuration file to use for obtaining the administrative username and password. Use this option, instead of the `username` and `password` attributes, in your build file when you do not want to have the plain text password shown in-line or in process-level utilities such as `ps`. Before specifying the `userconfigfile` attribute, you must first generate the file using the `weblogic.Admin STOREUSERCONFIG` command as described in STOREUSERCONFIG in the *WebLogic Server Command Reference*.	File	No
userkeyfile	Specifies the location of a user key file to use for encrypting and decrypting the username and password information stored in a user configuration file (the `userconfigfile` attribute). Before specifying the `userkeyfile` attribute, you must first generate the key file using the `weblogic.Admin STOREUSERCONFIG` command as described in STOREUSERCONFIG in the *WebLogic Server Command Reference*.	File	No

Nested Elements

`wlconfig` also has several elements that can be nested to specify configuration options:

- create
- delete
- set
- get
- query
- invoke

create

The `create` element creates a new MBean in the WebLogic Server domain. The `wlconfig` task can have any number of `create` elements.

A `create` element can have any number of nested `set` elements, which set attributes on the newly-created MBean. A `create` element may also have additional, nested `create` elements that create child MBeans.

The `create` element has the following attributes.

Table 2-3 Attributes of the create Element

Attribute	Description	Data Type	Required?
name	The name of the new MBean object to create.	String	No (`wlconfig` supplies a default name if none is specified.)
type	The MBean type.	String	Yes
property	The name of an optional Ant property that holds the object name of the newly-created MBean. **Note:** If you nest a `create` element inside of another `create` element, you cannot specify the `property` attribute for the *nested* `create` element.	String	No

delete

The `delete` element removes an existing MBean from the WebLogic Server domain. `delete` takes a single attribute:

Table 2-4 Attribute of the delete Element

Attribute	Description	Data Type	Required?
mbean	The object name of the MBean to delete.	String	Required when the `delete` element is a direct child of the `wlconfig` task. Not required when nested within a `query` element.

set

The `set` element sets MBean attributes on a named MBean, a newly-created MBean, or on MBeans retrieved as part of a query. You can include the `set` element as a direct child of the `wlconfig` task, or nested within a `create` or `query` element.

The `set` element has the following attributes:

Table 2-5 Attributes of the set Element

Attribute	Description	Data Type	Required?
attribute	The name of the MBean attribute to set.	String	Yes
value	The value to set for the specified MBean attribute. You can specify multiple object names (stored in Ant properties) as a value by delimiting the entire value list with quotes and separating the object names with a semicolon. See "Example of Setting Multiple Attribute Values" on page 2-14.	String	Yes

Table 2-5 Attributes of the set Element

Attribute	Description	Data Type	Required?
mbean	The object name of the MBean whose values are being set. This attribute is required only when the `set` element is included as a direct child of the main `wlconfig` task; it is not required when the set element is nested within the context of a `create` or `query` element.	String	Required only when the `set` element is a direct child of the `wlconfig` task.
domain	This attribute specifies the JMX domain name for Security MBeans and third-party SPI MBeans. It is not required for administration MBeans, as the domain corresponds to the WebLogic Server domain. **Note:** You cannot use this attribute if the `set` element is nested inside of a `create` element.	String	No

get

The `get` element retrieves attribute values from an MBean in the WebLogic Server domain. The `wlconfig` task can have any number of `get` elements.

The `get` element has the following attributes.

Table 2-6 Attributes of the get Element

Attribute	Description	Data Type	Required?
attribute	The name of the MBean attribute whose value you want to retrieve.	String	Yes

Table 2-6 Attributes of the get Element

Attribute	Description	Data Type	Required?
property	The name of an Ant property that will hold the retrieved MBean attribute value.	String	Yes
mbean	The object name of the MBean you want to retrieve attribute values from.	String	Yes

query

The `query` elements finds MBean that match a search pattern.

The query element supports the following nested child elements:

- `set`—performs set operations on all MBeans in the result set.
- `get`—performs get operations on all MBeans in the result set.
- `create`—each MBean in the result set is used as a parent of a new MBean.
- `delete`—performs delete operations on all MBeans in the result set.
- `invoke`—invokes all matching MBeans in the result set.

`wlconfig` can have any number of nested `query` elements.

`query` has the following attributes:

Table 2-7 Attributes of the query Element

Attribute	Description	Data Type	Required?
domain	The name of the WebLogic Server domain in which to search for MBeans.	String	No
type	The type of MBean to query.	String	No
name	The name of the MBean to query.	String	No
pattern	A JMX query pattern.	String	No

Table 2-7 Attributes of the query Element

Attribute	Description	Data Type	Required?
property	The name of an optional Ant property that will store the query results.	String	No
domain	This attribute specifies the JMX domain name for Security MBeans and third-party SPI MBeans. It is not required for administration MBeans, as the domain corresponds to the WebLogic Server domain.	String	No

invoke

The `invoke` element invokes a management operation for one or more MBeans. For WebLogic Server MBeans, you usually use this command to invoke operations other than the `get`*`Attribute`* and `set`*`Attribute`* that most WebLogic Server MBeans provide.

The `invoke` element has the following attributes.

Table 2-8 Attributes of the invoke Element

Attribute	Description	Data Type	Required?
mbean	The object name of the MBean you want to invoke.	String	You must specify either the `mbean` or `type` attribute of the invoke element.
type	The type of MBean to invoke.	String	You must specify either the `mbean` or `type` attribute of the invoke element.

Table 2-8 Attributes of the invoke Element

Attribute	Description	Data Type	Required?
methodName	The method of the MBean to invoke.	String	Yes
arguments	The list of arguments (separated by spaces) to pass to the method specified by the `methodName` attribute.	String	No

Using the libclasspath Ant Task

Use the `libclasspath` Ant task to build applications that use libraries, such as application libraries and web libraries.

- "libclasspath Task Definition" on page 2-22
- "wlserver Ant Task Reference" on page 2-4
- "Example libclasspath Ant Task" on page 2-23

libclasspath Task Definition

To use the task with your own Ant installation, add the following task definition in your build file:

```
   <taskdef name="libclasspath" classname="weblogic.ant.taskdefs.build.Lib
ClasspathTask"/>
```

libclasspath Ant Task Reference

The following sections describe the attributes and elements that can be used with the `libclasspath` Ant task.

- "Main libclasspath Attributes" on page 2-22
- "Nested libclasspath Elements" on page 2-23

Main libclasspath Attributes

The following table describes the main attributes of the `libclasspath` Ant task.

Table 2-9 Attributes of the libclasspath Ant Task

Attribute	Description	Required
`basedir`	The root of `.ear` or `.war` file to extract from.	One of the two attributes is required. If `basewar` is specified, `basedir` is ignored and the library referenced in `basewar` is used as the `.war` file to extract classpath or resourcepath information from.
`basewar`	The name of the .war file to extract from.	
`tmpdir`	The fully qualified name of the directory to be used for extracting libraries.	Yes.
`property`	Contains the classpath for the referenced libraries.	Yes.

Nested libclasspath Elements

`libclasspath` also has two elements that can be nested to specify configuration options. At least one of the elements is required when using the `libclasspath` Ant task:

librarydir

The following attribute is required when using this element:

`dir`—Specifies that all files in this directory are registered as available libraries.

library

The following attribute is required when using this element:

`file`—Register this file as an available library.

Example libclasspath Ant Task

This section provides example code of a libclasspath Ant task:

Listing 2-1 Example libclasspath Ant Task Code

```
.
.
.
   <taskdef name="libclasspath" classname="weblogic.ant.taskdefs.build.Lib
ClasspathTask"/>

   <!-- Builds classpath based on libraries defined in weblogic-applicatio
n.xml. -->
   <target name="init.app.libs">
      <libclasspath basedir="${src.dir}" tmpdir="${tmp.dir}" property="app
.lib.classpath">
         <librarydir dir="${weblogic.home}/common/deployable-libraries/"/>
      </libclasspath>
   <echo message="app.lib.claspath is ${app.lib.classpath}" level="info"/>
   </target>
   .
   .
   .
```

CHAPTER 3

Creating a Split Development Directory Environment

The following sections describe the steps for creating a WebLogic Server split development directory that you can use to develop a J2EE application or module:

Overview of the Split Development Directory Environment

The WebLogic split development directory environment consists of a directory layout and associated Ant tasks that help you repeatedly build, change, and deploy J2EE applications. Compared to other development frameworks, the WebLogic split development directory provides these benefits:

- **Fast development and deployment.** By minimizing unnecessary file copying, the split development directory Ant tasks help you recompile and redeploy applications quickly *without* first generating a deployable archive file or exploded archive directory.
- **Simplified build scripts.** The BEA-provided Ant tasks automatically determine which J2EE modules and classes you are creating, and build components in the correct order to support common classpath dependencies. In many cases, your project build script can simply identify the source and build directories and allow Ant tasks to perform their default behaviors.
- **Easy integration with source control systems.** The split development directory provides a clean separation between source files and generated files. This helps you maintain only editable files in your source control system. You can also clean the build by deleting the entire build directory; build files are easily replaced by rebuilding the project.

Source and Build Directories

The source and build directories form the basis of the split development directory environment. The source directory contains all editable files for your project—Java source files, editable descriptor files, JSPs, static content, and so forth. You create the source directory for an application by following the directory structure guidelines described in "Organizing J2EE Components in a Split Development Directory" on page 3-6.

The top level of the source directory always represents an Enterprise Application (`.ear` file), even if you are developing only a single J2EE module. Subdirectories beneath the top level source directory contain:

- Enterprise Application Modules (EJBs and Web Applications)

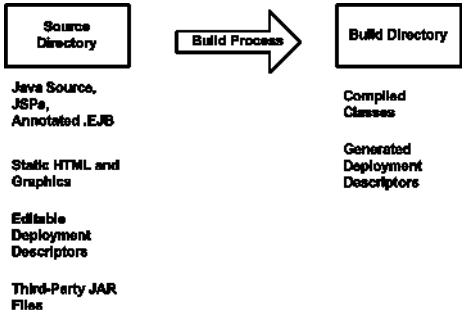
  Note: The split development directory structure does not provide support for developing new Resource Adapter components.

- Descriptor files for the Enterprise Application (`application.xml` and `weblogic-application.xml`)
- Utility classes shared by modules of the application (for example, exceptions, constants)

- Libraries (compiled `.jar` files, including third-party libraries) used by modules of the application

The build directory contents are generated automatically when you run the `wlcompile` ant task against a valid source directory. The `wlcompile` task recognizes EJB, Web Application, and shared library and class directories in the source directory, and builds those components in an order that supports common class path requirements. Additional Ant tasks can be used to build Web Services or generate deployment descriptor files from annotated EJB code.

Figure 3-1 Source and Build Directories



The build directory contains only those files generated during the build process. The combination of files in the source and build directories form a deployable J2EE application.

The build and source directory contents can be place in any directory of your choice. However, for ease of use, the directories are commonly placed in directories named `source` and `build`, within a single project directory (for example, `\myproject\build` and `\myproject\source`).

Deploying from a Split Development Directory

All WebLogic Server deployment tools (`weblogic.Deployer`, `wldeploy`, and the Administration Console) support direct deployment from a split development directory. You specify only the build directory when deploying the application to WebLogic Server.

WebLogic Server attempts to use all classes and resources available in the *source* directory for deploying the application. If a required resource is not available in the source directory,

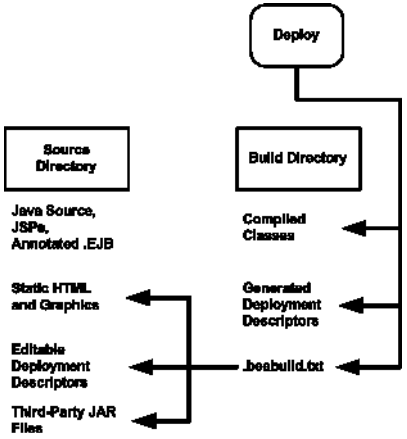
WebLogic Server then looks in the application's build directory for that resource. For example, if a deployment descriptor is generated during the build process, rather than stored with source code as an editable file, WebLogic Server obtains the generated file from the build directory.

WebLogic Server discovers the location of the source directory by examining the `.beabuild.txt` file that resides in the top level of the application's build directory. If you ever move or modify the source directory location, edit the `.beabuild.txt` file to identify the new source directory name.

"Deploying and Packaging from a Split Development Directory" on page 5-1 describes the `wldeploy` Ant task that you can use to automate deployment from the split directory environment.

Figure 3-2 shows a typical deployment process. The process is initiated by specifying the build directory with a WebLogic Server tool. In the figure, all compiled classes and generated deployment descriptors are discovered in the build directory, but other application resources (such as static files and editable deployment descriptors) are missing. WebLogic Server uses the hidden `.beabuild.txt` file to locate the application's source directory, where it finds the required resources.

Figure 3-2 Split Directory Deployment



Split Development Directory Ant Tasks

BEA provides a collection of Ant tasks designed to help you develop applications using the split development directory environment. Each Ant task uses the source, build, or both directories to perform common development tasks:

- `wlcompile`—This Ant task compiles the contents of the source directory into subdirectories of the build directory. `wlcompile` compiles Java classes and also processes annotated `.ejb` files into deployment descriptors, as described in "Compiling Applications Using wlcompile" on page 4-1.
- `wlappc`—This Ant task invokes the appc compiler, which generates JSPs and container-specific EJB classes for deployment. See "Building Modules and Applications Using wlappc" on page 4-4.
- `wldeploy`—This Ant task deploys any format of J2EE applications (exploded or archived) to WebLogic Server. To deploy directly from the split development directory environment, you specify the build directory of your application. See "wldeploy Ant Task Reference" on page B-1.
- `wlpackage`—This Ant task uses the contents of both the source and build directories to generate an EAR file or exploded EAR directory that you can give to others for deployment.

Using the Split Development Directory Structure: Main Steps

The following steps illustrate how you use the split development directory structure to build and deploy a WebLogic Server application.

1. Create the main EAR source directory for your project. When using the split development directory environment, you must develop Web Applications and EJBs as part of an Enterprise Application, even if you do not intend to develop multiple J2EE modules. See "Organizing J2EE Components in a Split Development Directory" on page 3-6.
2. Add one or more subdirectories to the EAR directory for storing the source for Web Applications, EJB components, or shared utility classes. See "Organizing J2EE Components in a Split Development Directory" on page 3-6 and "Organizing Shared Classes in a Split Development Directory" on page 3-12.
3. Store all of your editable files (source code, static content, editable deployment descriptors) for modules in subdirectories of the EAR directory. Add the entire contents of the source directory to your source control system, if applicable.

4. Set your WebLogic Server environment by executing either the `setWLSEnv.cmd` (Windows) or `setWLSEnv.sh` (UNIX) script. The scripts are located in the *`WL_HOME`*`\server\bin\` directory, where *`WL_HOME`* is the top-level directory in which WebLogic Server is installed.
5. Use the `weblogic.BuildXMLGen` utility to generate a default `build.xml` file for use with your project. Edit the default property values as needed for your environment. See "Generating a Basic build.xml File Using weblogic.BuildXMLGen" on page 3-13.
6. Use the default targets in the `build.xml` file to build, deploy, and package your application. See "Generating a Basic build.xml File Using weblogic.BuildXMLGen" on page 3-13 for a list of default targets.

Organizing J2EE Components in a Split Development Directory

The split development directory structure requires each project to be staged as a J2EE Enterprise Application. BEA therefore recommends that you stage even stand-alone Web applications and EJBs as modules of an Enterprise application, to benefit from the split directory Ant tasks. This practice also allows you to easily add or remove modules at a later date, because the application is already organized as an EAR.

Note: If your project requires multiple EARs, see also "Developing Multiple-EAR Projects Using the Split Development Directory" on page 3-15.

The following sections describe the basic conventions for staging the following module types in the split development directory structure:

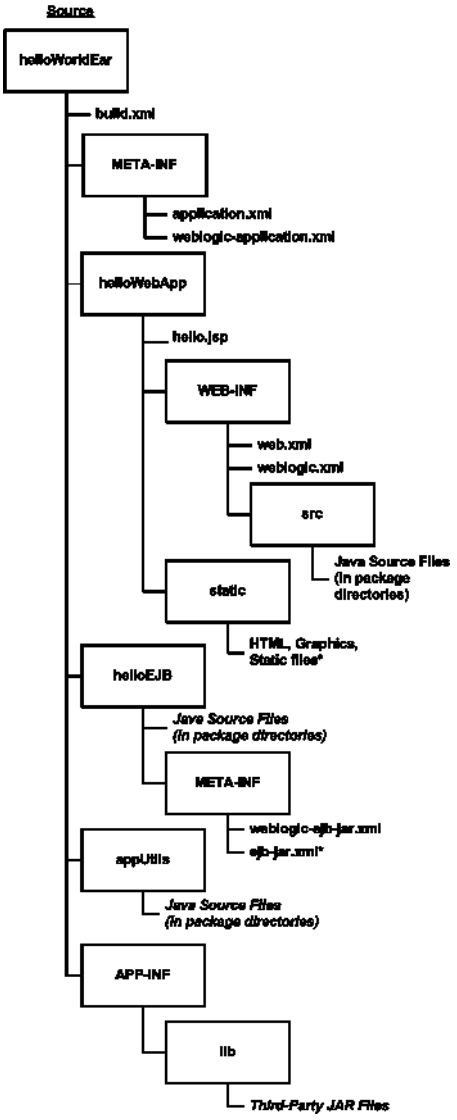
- "Enterprise Application Configuration" on page 3-9
- "Web Applications" on page 3-9
- "EJBs" on page 3-11
- "Shared Utility Classes" on page 3-12
- "Third-Party Libraries" on page 3-13

The directory examples are taken from the `splitdir` sample application installed in *`WL_HOME`*`\samples\server\examples\src\examples\splitdir`, where *`WL_HOME`* is your WebLogic Server installation directory.

Source Directory Overview

The following figure summarizes the source directory contents of an Enterprise Application having a Web Application, EJB, shared utility classes, and third-party libraries. The sections that follow provide more details about how individual parts of the enterprise source directory are organized.

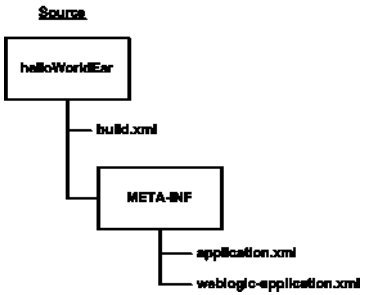
Figure 3-3 Overview of Enterprise Application Source Directory



Enterprise Application Configuration

The top level source directory for a split development directory project represents an Enterprise Application. The following figure shows the minimal files and directories required in this directory.

Figure 3-4 Enterprise Application Source Directory

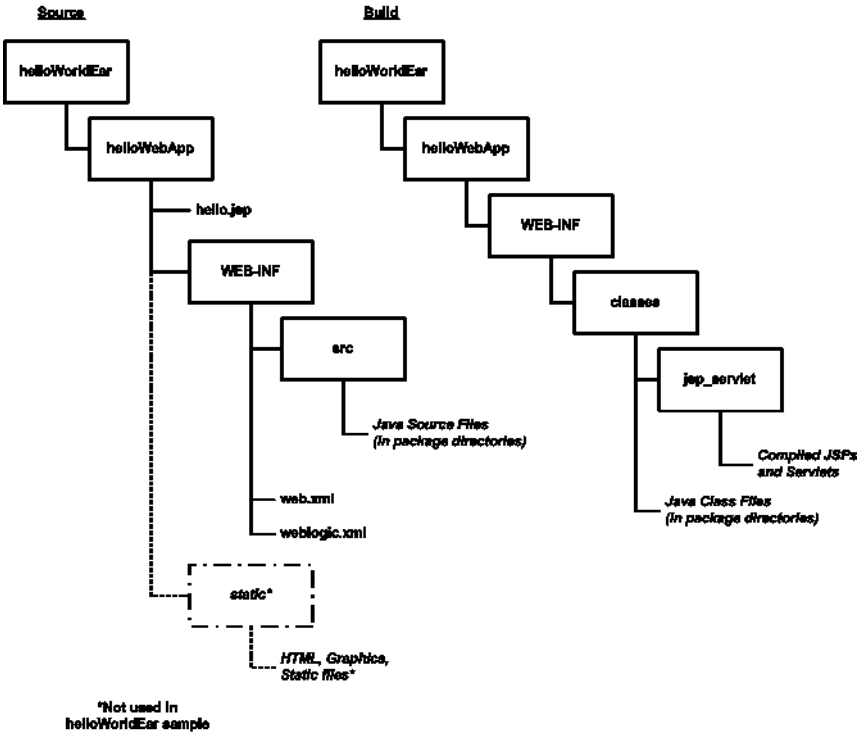


The Enterprise Application directory will also have one or more subdirectories to hold a Web Application, EJB, utility class, and/or third-party Jar file, as described in the following sections.

Web Applications

Web Applications use the basic source directory layout shown in the figure below.

Figure 3-5 Web Application Source and Build Directories



The key directories and files for the Web Application are:

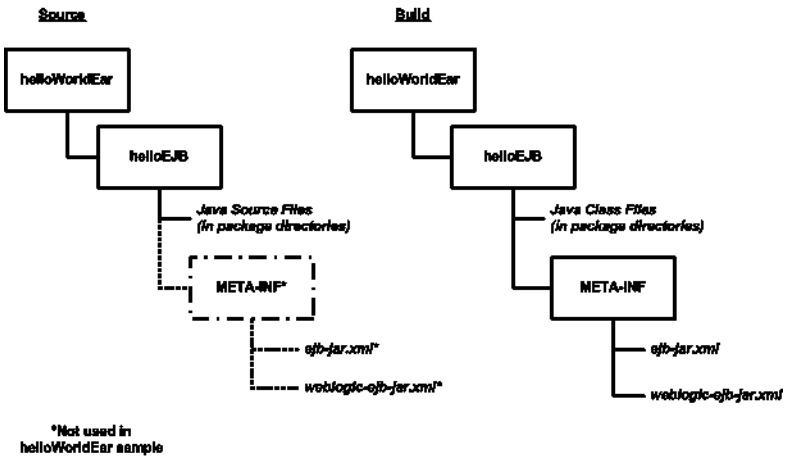
- `helloWebApp\` —The top level of the Web Application module can contain JSP files and static content such as HTML files and graphics used in the application. You can also store static files in any named subdirectory of the Web Application (for example, `helloWebApp\graphics` or `helloWebApp\static`.)
- `helloWebApp\WEB-INF\` —Store the Web Application's editable deployment descriptor files (`web.xml` and `weblogic.xml`) in the `WEB-INF` subdirectory.
- `helloWebApp\WEB-INF\src` —Store Java source files for Servlets in package subdirectories under `WEB-INF\src`.

When you build a Web Application, the `appc` Ant task and `jspc` compiler compile JSPs into package subdirectories under `helloWebApp\WEB-INF\classes\jsp_servlet` in the build directory. Editable deployment descriptors are not copied during the build process.

EJBs

EJBs use the source directory layout shown in the figure below.

Figure 3-6 EJB Source and Build Directories



The key directories and files for an EJB are:

- `helloEJB\` —Store all EJB source files under package directories of the EJB module directory. The source files can be either `.java` source files, or annotated `.ejb` files.
- `helloEJB\META-INF\` —Store editable EJB deployment descriptors (ejb-jar.xml and weblogic-ejb-jar.xml) in the `META-INF` subdirectory of the EJB module directory. The `helloWorldEar` sample does not include a `helloEJB\META-INF` subdirectory, because its deployment descriptors files are generated from annotations in the `.ejb` source files. See "Important Notes Regarding EJB Descriptors" on page 3-11.

During the build process, EJB classes are compiled into package subdirectories of the `helloEJB` module in the build directory. If you use annotated `.ejb` source files, the build process also generates the EJB deployment descriptors and stores them in the `helloEJB\META-INF` subdirectory of the build directory.

Important Notes Regarding EJB Descriptors

EJB deployment descriptors should be included in the source `META-INF` directory and treated as source code *only* if those descriptor files are created from scratch or are edited manually.

Descriptor files that are generated from annotated `.ejb` files should appear only in the build directory, and they can be deleted and regenerated by building the application.

For a given EJB component, the EJB source directory should contain either:

- EJB source code in `.java` source files and editable deployment descriptors in `META-INF`

or:

- EJB source code with descriptor annotations in `.ejb` source files, and *no editable descriptors* in `META-INF`.

In other words, do not provide both annotated `.ejb` source files and editable descriptor files for the same EJB component.

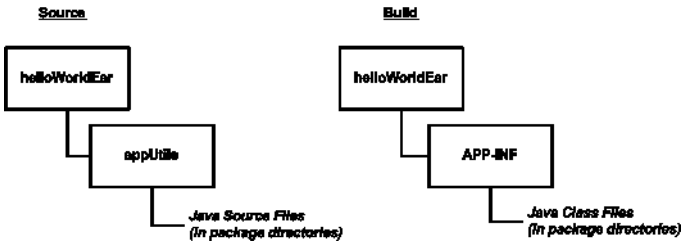
Organizing Shared Classes in a Split Development Directory

The WebLogic split development directory also helps you store shared utility classes and libraries that are required by modules in your Enterprise Application. The following sections describe the directory layout and classloading behavior for shared utility classes and third-party JAR files.

Shared Utility Classes

Enterprise Applications frequently use Java utility classes that are shared among application modules. Java utility classes differ from third-party JARs in that the source files are part of the application and must be compiled. Java utility classes are typically libraries used by application modules such as EJBs or Web applications.

Figure 3-7 Java Utility Class Directory



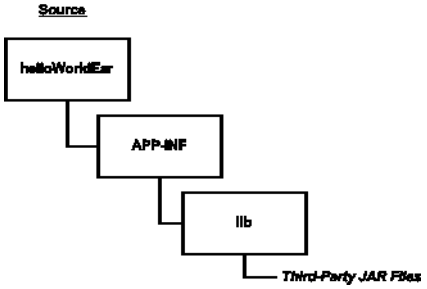
Place the source for Java utility classes in a named subdirectory of the top-level Enterprise Application directory. Beneath the named subdirectory, use standard package subdirectory conventions.

During the build process, the `wlcompile` Ant task invokes the `javac` compiler and compiles Java classes into the `APP-INF/classes/` directory under the build directory. This ensures that the classes are available to other modules in the deployed application.

Third-Party Libraries

You can extend an Enterprise Application to use third-party `.jar` files by placing the files in the `APP-INF\lib\` directory, as shown below:

Figure 3-8 Third-party Library Directory



Third-party JARs are generally not compiled, but may be versioned using the source control system for your application code. For example, XML parsers, logging implementations, and Web Application framework JAR files are commonly used in applications and maintained along with editable source code.

During the build process, third-party JAR files are not copied to the build directory, but remain in the source directory for deployment.

Class Loading for Shared Classes

The classes and libraries stored under `APP-INF/classes` and `APP-INF/lib` are available to all modules in the Enterprise Application. The application classloader always attempts to resolve class requests by first looking in `APP-INF/classes`, then `APP-INF/lib`.

Generating a Basic build.xml File Using weblogic.BuildXMLGen

After you set up your source directory structure, use the `weblogic.BuildXMLGen` utility to create a basic `build.xml` file. `weblogic.BuildXMLGen` is a convenient utility that generates an Ant `build.xml` file for Enterprise applications that are organized in the split

development directory structure. The utility analyzes the source directory and creates build and deploy targets for the Enterprise application as well as individual modules. It also creates targets to clean the build and generate new deployment descriptors.

The syntax for `weblogic.BuildXMLGen` is as follows:

```
java weblogic.BuildXMLGen [options] <source directory>
```

where `options` include:

- -help—print standard usage message
- -version—print version information
- `-projectName <project name>`—name of the Ant project
- `-d <directory>`—directory where `build.xml` is created. The default is the current directory.
- `-file <build.xml>`—name of the generated build file
- `-librarydir <directories>`—create build targets for shared J2EE libraries in the comma-separated list of directories. See "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.
- `-username <username>`—user name for deploy commands
- `-password <password>`—user password

After running `weblogic.BuildXMLGen`, edit the generated `build.xml` file to specify properties for your development environment. The list of properties you need to edit are shown in the listing below.

Listing 3-1 build.xml Editable Properties

```
<!-- BUILD PROPERTIES ADJUST THESE FOR YOUR ENVIRONMENT -->
  <property name="tmp.dir" value="/tmp" />
  <property name="dist.dir" value="${tmp.dir}/dist"/>
  <property name="app.name" value="helloWorldEar" />
  <property name="ear" value="${dist.dir}/${app.name}.ear"/>
  <property name="ear.exploded" value="${dist.dir}/${app.name}_exploded"/>
  <property name="verbose" value="true" />
  <property name="user" value="USERNAME" />
```

```
<property name="password" value="PASSWORD" />
<property name="servername" value="myserver" />
<property name="adminurl" value="iiop://localhost:7001" />
```

In particular, make sure you edit the `tmp.dir` property to point to the build directory you want to use. By default, the `build.xml` file builds projects into a subdirectory `tmp.dir` named after the application (`/tmp/helloWorldEar` in the above listing).

The following listing shows the default main targets created in the `build.xml` file. You can view these targets at the command prompt by entering the `ant -projecthelp` command in the EAR source directory.

Listing 3-2 Default build.xml Targets

```
appc                 Runs weblogic.appc on your application
build                Compiles helloWorldEar application and runs appc
clean                Deletes the build and distribution directories
compile              Only compiles helloWorldEar application, no appc
compile.appStartup   Compiles just the appStartup module of the application
compile.appUtils     Compiles just the appUtils module of the application
compile.build.orig   Compiles just the build.orig module of the application
compile.helloEJB     Compiles just the helloEJB module of the application
compile.helloWebApp  Compiles just the helloWebApp module of the application
compile.javadoc      Compiles just the javadoc module of the application
deploy              Deploys (and redeploys) the entire helloWorldEar application
descriptors          Generates application and module descriptors
ear                  Package a standard J2EE EAR for distribution
ear.exploded         Package a standard exploded J2EE EAR
redeploy.appStartup  Redeploys just the appStartup module of the application
redeploy.appUtils    Redeploys just the appUtils module of the application
redeploy.build.orig  Redeploys just the build.orig module of the application
redeploy.helloEJB    Redeploys just the helloEJB module of the application
redeploy.helloWebApp Redeploys just the helloWebApp module of application
redeploy.javadoc     Redeploys just the javadoc module of the application
undeploy               Undeploys the entire helloWorldEar application
```

Developing Multiple-EAR Projects Using the Split Development Directory

The split development directory examples and procedures described previously have dealt with projects consisting of a single Enterprise Application. Projects that require building multiple Enterprise Applications simultaneously require slightly different conventions and procedures, as described in the following sections.

Note: The following sections refer to the MedRec sample application, which consists of three separate Enterprise Applications as well as shared utility classes, third-party JAR files, and dedicated client applications. The MedRec source and build directories are installed under *WL_HOME*/samples/server/medrec, where *WL_HOME* is the WebLogic Server installation directory.

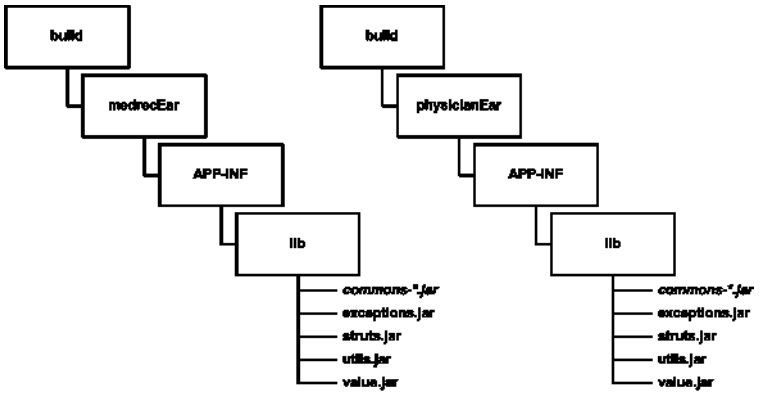
Organizing Libraries and Classes Shared by Multiple EARs

For single EAR projects, the split development directory conventions suggest keeping third-party JAR files in the APP-INF/lib directory of the EAR source directory. However, a multiple-EAR project would require you to maintain a copy of the same third-party JAR files in the APP-INF/lib directory of *each* EAR source directory. This introduces multiple copies of the source JAR files, increases the possibility of some JAR files being at different versions, and requires additional space in your source control system.

To address these problems, consider editing your build script to copy third-party JAR files into the APP-INF/lib directory of the *build* directory for each EAR that requires the libraries. This allows you to maintain a single copy and version of the JAR files in your source control system, yet it enables each EAR in your project to use the JAR files.

The MedRec sample application installed with WebLogic Server uses this strategy, as shown in the following figure.

Figure 3-9 Shared JAR Files in MedRec



MedRec takes a similar approach to utility classes that are shared by multiple EARs in the project. Instead of including the source for utility classes within the scope of each ear that needs them, MedRec keeps the utility class source independent of all EARs. After compiling the utility

classes, the build script archives them and copies the JARs into the build directory under the `APP-INF/LIB` subdirectory of each EAR that uses the classes, as shown in figure Figure 3-9.

Linking Multiple build.xml Files

When developing multiple EARs using the split development directory, each EAR project generally uses its own `build.xml` file (perhaps generated by multiple runs of `weblogic.BuildXMLGen`.). Applications like MedRec also use a master `build.xml` file that calls the subordinate `build.xml` files for each EAR in the application suite.

Ant provides a core task (named `ant`) that allows you to execute other project build files within a master `build.xml` file. The following line from the MedRec master build file shows its usage:

```
<ant inheritAll="false" dir="${root}/startupEar" antfile="build.xml"/>
```

The above task instructs Ant to execute the file named `build.xml` in the `/startupEar` subdirectory. The `inheritAll` parameter instructs Ant to pass only user properties from the master build file tot the `build.xml` file in `/startupEar`.

MedRec uses multiple tasks similar to the above to build the `startupEar`, `medrecEar`, and `physicianEar` applications, as well as building common utility classes and client applications.

Best Practices for Developing WebLogic Server Applications

BEA recommends the following "best practices" for application development.

- Package applications as part of an Enterprise application. See "Packaging Applications Using wlpackage" on page 5-2.
- Use the split development directory structure. See "Organizing J2EE Components in a Split Development Directory" on page 3-6.
- For distribution purposes, package and deploy in archived format. See "Packaging Applications Using wlpackage" on page 5-2.
- In most other cases, it is more convenient to deploy in exploded format. See "Archive versus Exploded Archive Directory" on page 5-2.
- Never deploy untested code on a WebLogic Server instance that is serving production applications. Instead, set up a development WebLogic Server instance on the same computer on which you edit and compile, or designate a WebLogic Server development location elsewhere on the network.

- Even if you do not run a development WebLogic Server instance on your development computer, you must have access to a WebLogic Server distribution to compile your programs. To compile any code using WebLogic or J2EE APIs, the Java compiler needs access to the `weblogic.jar` file and other JAR files in the distribution directory. Install WebLogic Server on your development computer to make WebLogic distribution files available locally.

CHAPTER 4

Building Applications in a Split Development Directory

The following sections describe the steps for building WebLogic Server J2EE applications using the WebLogic split development directory environment:

Compiling Applications Using wlcompile

You use the `wlcompile` Ant task to invoke the `javac` compiler to compile your application's Java components in a split development directory structure. The basic syntax of `wlcompile` identifies the source and build directories, as in this command from the `helloWorldEar` sample:

```
<wlcompile srcdir="${src.dir}" destdir="${dest.dir}"/>
```

The following is the order in which events occur using this task:

1. `wlcompile` compiles the Java components into an output directory:

   WL_HOME\samples\server\examples\build\helloWorldEar\APP-INF\classes\

   where *WL_HOME* is the WebLogic Server installation directory.

2. `wlcompile` builds the EJBs and automatically includes the previously built Java modules in the compiler's classpath. This allows the EJBs to call the Java modules without requiring you to manually edit their classpath.

3. Finally, `wlcompile` compiles the Java components in the Web application with the EJB and Java modules in the compiler's classpath. This allows the Web applications to refer to the EJB and application Java classes without requiring you to manually edit the classpath.

Using includes and excludes Properties

More complex Enterprise applications may have compilation dependencies that are not automatically handled by the `wlcompile` task. However, you can use the include and exclude options to `wlcompile` to enforce your own dependencies. The `includes` and `excludes` properties accept the names of Enterprise Application modules—the names of subdirectories in the Enterprise application source directory—to include or exclude them from the compile stage.

The following line from the `helloWorldEar` sample shows the `appStartup` module being excluded from compilation:

```
<wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
   excludes="appStartup"/>
```

wlcompile Ant Task Attributes

Table 4-1 contains Ant task attributes specific to `wlcompile`.

Table 4-1 wlcompile Ant Task Attributes

Attribute	Description
srcdir	The source directory.
destdir	The build/output directory.
classpath	Allows you to change the classpath used by `wlcompile`.
includes	Allows you to include specific directories from the build.
excludes	Allows you to exclude specific directories from the build.
librarydir	Specifies a directory of shared J2EE libraries to add to the classpath. See "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.

Nested javac Options

The `wlcompile` Ant task can accept nested `javac` options to change the compile-time behavior. For example, the following `wlcompile` command ignores deprecation warnings and enables debugging:

```
<wlcompile srcdir="${mysrcdir}" destdir="${mybuilddir}">
   <javac deprecation="false" debug="true"
   debuglevel="lines,vars,source"/>
</wlcompile>
```

Setting the Classpath for Compiling Code

Most WebLogic services are based on J2EE standards and are accessed through standard J2EE packages. The Sun, WebLogic, and other Java classes required to compile programs that use WebLogic services are packaged in the `weblogic.jar` file in the `lib` directory of your WebLogic Server installation. In addition to `weblogic.jar`, include the following in your compiler's `CLASSPATH`:

- The `lib\tools.jar` file in the JDK directory, or other standard Java classes required by the Java Development Kit you use.
- The `examples.property` file for Apache Ant (for examples environment). This file is discussed in the WebLogic Server documentation on building examples using Ant located at: `samples\server\examples\src\examples\examples.html`
- Classes for third-party Java tools or services your programs import.
- Other application classes referenced by the programs you are compiling.

Library Element for wlcompile and wlappc

The `library` element is an optional element used to define the name and optional version information for a module that represents a shared J2EE library required for building an application, as described in "Creating Shared J2EE Libraries and Optional Packages" on page 8-1. The `library` element can be used with both `wlcompile` and `wlappc`, described in "Building Modules and Applications Using wlappc" on page 4-4.

The name and version information are specified as attributes to the library element, described in "Library attributes" on page 4-4

Table 4-2 Library attributes

Attribute	Description
file	Required filename of a J2EE library
name	The optional name of a required J2EE library.
specificationversion	An optional specification version required for the library.
implementationversion	An optional implementation version required for the library.

The format choices for both `specificationversion` and `implementationversion` are described in "Referencing Shared J2EE Libraries in an Enterprise Application" on page 8-11. The following output shows a sample `library` reference:

```
<library file="c:\mylibs\lib.jar" name="ReqLib"
specificationversion="90Beta" implementationversion="1.1" />
```

Building Modules and Applications Using wlappc

The `weblogic.appc` compiler generates JSPs and container-specific EJB classes for deployment, and validates deployment descriptors for compliance with the current J2EE specifications. `appc` performs validation checks between the application-level deployment descriptors and the individual modules in the application as well as validation checks across the modules.

`wlappc` is the Ant task interface to the `weblogic.appc` compiler. The following section describe the `wlappc` options and usage.

Both `weblogic.appc` and the `wlappc` Ant task compile modules in the order in which they appear in the `application.xml` deployment descriptor file that describes your Enterprise application.

wlappc Ant Task Attributes

Table 4-3 describes Ant task options specific to `wlappc`. These options are similar to the `weblogic.appc` command-line options, but with a few differences.

Notes: See "weblogic.appc Reference" on page 4-7 for a list of `weblogic.appc` options.

See also "Library Element for wlcompile and wlappc" on page 4-3.

Table 4-3 wlappc Ant Task Attributes

Option	Description
`print`	Prints the standard usage message.
`version`	Prints `appc` version information.
`output <file>`	Specifies an alternate output archive or directory. If not set, the output is placed in the source archive or directory.
`forceGeneration`	Forces generation of EJB and JSP classes. Without this flag, the classes may not be regenerated (if determined to be unnecessary).
`lineNumbers`	Adds line numbers to generated class files to aid in debugging.
basicClientJar	Does not include deployment descriptors in client JARs generated for EJBs.
idl	Generates IDL for EJB remote interfaces.
idlOverwrite	Always overwrites existing IDL files.
idlVerbose	Displays verbose information for IDL generation.
idlNoValueTypes	Does not generate valuetypes and the methods/attributes that contain them.
idlNoAbstractInterfaces	Does not generate abstract interfaces and methods/attributes that contain them.
idlFactories	Generates factory methods for valuetypes.
idlVisibroker	Generates IDL somewhat compatible with Visibroker 4.5 C++.
idlOrbix	Generates IDL somewhat compatible with Orbix 2000 2.0 C++.
idlDirectory <dir>	Specifies the directory where IDL files will be created (default: target directory or JAR)
idlMethodSignatures <>	Specifies the method signatures used to trigger IDL code generation.
iiop	Generates CORBA stubs for EJBs.

iiopDirectory <dir>	Specifies the directory where IIOP stub files will be written (default: target directory or JAR)
keepgenerated	Keeps the generated `.java` files.
librarydir	Specifies a directory of shared J2EE libraries to add to the classpath. See "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.
compiler <javac>	Selects the Java compiler to use.
`debug`	Compiles debugging information into a class file.
`optimize`	Compiles with optimization on.
nowarn	Compiles without warnings.
verbose	Compiles with verbose output.
deprecation	Warns about deprecated calls.
normi	Passes flags through to Symantec's sj.
`runtimeflags`	Passes flags through to Java runtime
classpath <path>	Selects the classpath to use during compilation.
advanced	Prints advanced usage options.

wlappc Ant Task Syntax

The basic syntax for using the `wlappc` Ant task determines the destination source directory location. This directory contains the files to be compiled by `wlappc`.

```
<wlappc source="${dest.dir}" />
```

The following is an example of a `wlappc` Ant task command that invokes two options (`idl` and `idlOrverWrite`) from Table 4-3.

```
<wlappc source="${dest.dir}"idl="true" idlOrverWrite="true" />
```

Syntax Differences between appc and wlappc

There are some syntax differences between `appc` and `wlappc`. For appc, the presence of a flag in the command is a boolean. For `wlappc`, the presence of a flag in the command means that the argument is required.

To illustrate, the following are examples of the same command, the first being an `appc` command and the second being a `wlappc` command:

```
java weblogic.appc -idl foo.ear
```

```
<wlappc source="${dest.dir} idl="true"/>
```

weblogic.appc Reference

The following sections describe how to use the command-line version of the `appc` compiler. The `weblogic.appc` command-line compiler reports any warnings or errors encountered in the descriptors and compiles all of the relevant modules into an EAR file, which can be deployed to WebLogic Server.

weblogic.appc Syntax

Use the following syntax to run `appc`:

```
prompt>java weblogic.appc [options] <ear, jar, or war file or directory>
```

weblogic.appc Options

The following are the available `appc` options:

Table 4-4 appc Options:

Option	Description
`-print`	Prints the standard usage message.
`-version`	Prints `appc` version information.
`-output <file>`	Specifies an alternate output archive or directory. If not set, the output is placed in the source archive or directory.
`-forceGeneration`	Forces generation of EJB and JSP classes. Without this flag, the classes may not be regenerated (if determined to be unnecessary).

-library <file[[@name=<string>][@libspecver=<version>][@libimplver=<version|string>]]>	A comma-separated list of shared J2EE libraries. Optional name and version string information must be specified in the format described in "Referencing Shared J2EE Libraries in an Enterprise Application" on page 8-11.
`-lineNumbers`	Adds line numbers to generated class files to aid in debugging.
-basicClientJar	Does not include deployment descriptors in client JARs generated for EJBs.
-idl	Generates IDL for EJB remote interfaces.
-idlOverwrite	Always overwrites existing IDL files.
-idlVerbose	Displays verbose information for IDL generation.
-idlNoValueTypes	Does not generate valuetypes and the methods/attributes that contain them.
-idlNoAbstractInterfaces	Does not generate abstract interfaces and methods/attributes that contain them.
-idlFactories	Generates factory methods for valuetypes.
-idlVisibroker	Generates IDL somewhat compatible with Visibroker 4.5 C++.
-idlOrbix	Generates IDL somewhat compatible with Orbix 2000 2.0 C++.
-idlDirectory <dir>	Specifies the directory where IDL files will be created (default: target directory or JAR)
-idlMethodSignatures <>	Specifies the method signatures used to trigger IDL code generation.
-iiop	Generates CORBA stubs for EJBs.
-iiopDirectory <dir>	Specifies the directory where IIOP stub files will be written (default: target directory or JAR)
-keepgenerated	Keeps the generated `.java` files.
-compiler <javac>	Selects the Java compiler to use.
-g	Compiles debugging information into a class file.

-O	Compiles with optimization on.
-nowarn	Compiles without warnings.
-verbose	Compiles with verbose output.
-deprecation	Warns about deprecated calls.
-normi	Passes flags through to Symantec's sj.
-J<option>	Passes flags through to Java runtime.
-classpath <path>	Selects the classpath to use during compilation.
-advanced	Prints advanced usage options.

CHAPTER 5

Deploying and Packaging from a Split Development Directory

The following sections describe the steps for deploying WebLogic Server J2EE applications using the WebLogic split development directory environment:

Deploying Applications Using wldeploy

The `wldeploy` task provides an easy way to deploy directly from the split development directory. `wlcompile` provides most of the same arguments as the `weblogic.Deployer` directory. To deploy from a split development directory, you simply identify the build directory location as the deployable files, as in:

```
<wldeploy user="${user}" password="${password}"
       action="deploy" source="${dest.dir}"
       name="helloWorldEar" />
```

The above task is automatically created when you use `weblogic.BuildXMLGen` to create the `build.xml` file.

See "wldeploy Ant Task Reference" on page B-1 for a complete command reference.

Packaging Applications Using wlpackage

The `wlpackage` Ant task uses the contents of both the source and build directories to create either a deployable archive file (`.EAR` file), or an exploded archive directory representing the Enterprise Application (exploded `.EAR` directory). Use `wlpackage` when you want to deliver your application to another group or individual for evaluation, testing, performance profiling, or production deployment.

Archive versus Exploded Archive Directory

For production purposes, it is convenient to deploy Enterprise applications in exploded (unarchived) directory format. This applies also to stand-alone Web applications, EJBs, and connectors packaged as part of an Enterprise application. Using this format allows you to update files directly in the exploded directory rather than having to unarchive, edit, and rearchive the whole application. Using exploded archive directories also has other benefits, as described in Deployment Archive Files Versus Exploded Archive Directories in *Deploying Applications to WebLogic Server*.

You can also package applications in a single archived file, which is convenient for packaging modules and applications for distribution. Archive files are easier to copy, they use up fewer file handles than an exploded directory, and they can save disk space with file compression.

The Java classloader can search for Java class files (and other file types) in a JAR file the same way that it searches a directory in its classpath. Because the classloader can search a directory or a JAR file, you can deploy J2EE modules on WebLogic Server in either a JAR (archived) file or an exploded (unarchived) directory.

wlpackage Ant Task

In a production environment, use the `wlpackage` Ant task to package your split development directory application as a traditional EAR file that can be deployed to WebLogic Server. Continuing with the MedRec example, you would package your application as follows:

```
<wlpackage toFile="\physicianEAR\physicianEAR.ear" srcdir="\physicianEAR"

destdir="\build\physicianEAR"/>

<wlpackage toDir="\physicianEAR\explodedphysicianEar"
srcdir="\src\physicianEAR"

destdir="\build\physicianEAR" />
```

CHAPTER 6

Understanding WebLogic Server Application Classloading

The following sections provide an overview of Java classloaders, followed by details about WebLogic Server J2EE application classloading.

Java Classloader Overview

Classloaders are a fundamental module of the Java language. A classloader is a part of the Java virtual machine (JVM) that loads classes into memory; a classloader is responsible for finding and loading class files at run time. Every successful Java programmer needs to understand classloaders and their behavior. This section provides an overview of Java classloaders.

Java Classloader Hierarchy

Classloaders contain a hierarchy with parent classloaders and child classloaders. The relationship between parent and child classloaders is analogous to the object relationship of super classes and subclasses. The bootstrap classloader is the root of the Java classloader hierarchy. The Java virtual machine (JVM) creates the bootstrap classloader, which loads the Java development kit (JDK) internal classes and `java.*` packages included in the JVM. (For example, the bootstrap classloader loads `java.lang.String`.)

The extensions classloader is a child of the bootstrap classloader. The extensions classloader loads any JAR files placed in the extensions directory of the JDK. This is a convenient means to extending the JDK without adding entries to the classpath. However, anything in the extensions directory must be self-contained and can only refer to classes in the extensions directory or JDK classes.

The system classpath classloader extends the JDK extensions classloader. The system classpath classloader loads the classes from the classpath of the JVM. Application-specific classloaders (including WebLogic Server classloaders) are children of the system classpath classloader.

Note: What BEA refers to as a "system classpath classloader" is often referred to as the "application classloader" in contexts outside of WebLogic Server. When discussing classloaders in WebLogic Server, BEA uses the term "system" to differentiate from classloaders related to J2EE applications (which BEA refers to as "application classloaders").

Loading a Class

Classloaders use a delegation model when loading a class. The classloader implementation first checks its cache to see if the requested class has already been loaded. This class verification improves performance in that its cached memory copy is used instead of repeated loading of a class from disk. If the class is not found in its cache, the current classloader asks its parent for the class. Only if the parent cannot load the class does the classloader attempt to load the class. If a class exists in both the parent and child classloaders, the parent version is loaded. This delegation

model is followed to avoid multiple copies of the same form being loaded. Multiple copies of the same class can lead to a `ClassCastException`.

Classloaders ask their parent classloader to load a class before attempting to load the class themselves. Classloaders in WebLogic Server that are associated with Web applications can be configured to check locally first before asking their parent for the class. This allows Web applications to use their own versions of third-party classes, which might also be used as part of the WebLogic Server product. The "prefer-web-inf-classes Element" on page 6-3 section discusses this in more detail.

prefer-web-inf-classes Element

The `weblogic.xml` Web application deployment descriptor contains a `<prefer-web-inf-classes>` element (a sub-element of the `<container-descriptor>` element). By default, this element is set to `False`. Setting this element to `True` subverts the classloader delegation model so that class definitions from the Web application are loaded in preference to class definitions in higher-level classloaders. This allows a Web application to use its own version of a third-party class, which might also be part of WebLogic Server. See "weblogic.xml Deployment Descriptor Elements."

When using this feature, you must be careful not to mix instances created from the Web application's class definition with issuances created from the server's definition. If such instances are mixed, a `ClassCastException` results.

Listing 6-1 illustrates the `prefer-web-inf-classes` element, its description and default value.

Listing 6-1 prefer-web-inf-classes Element

```
/**
* If true, classes located in the WEB-INF directory of a web-app will be
* loaded in preference to classes loaded in the application or system
* classloader.
* @default false
*/
boolean isPreferWebInfClasses();
void setPreferWebInfClasses(boolean b);
```

Changing Classes in a Running Program

WebLogic Server allows you to deploy newer versions of application modules such as EJBs while the server is running. This process is known as hot-deploy or hot-redeploy and is closely related to classloading.

Java classloaders do not have any standard mechanism to undeploy or unload a set of classes, nor can they load new versions of classes. In order to make updates to classes in a running virtual machine, the classloader that loaded the changed classes must be replaced with a new classloader. When a classloader is replaced, all classes that were loaded from that classloader (or any classloaders that are offspring of that classloader) must be reloaded. Any instances of these classes must be re-instantiated.

In WebLogic Server, each application has a hierarchy of classloaders that are offspring of the system classloader. These hierarchies allow applications or parts of applications to be individually reloaded without affecting the rest of the system. "WebLogic Server Application Classloader Overview" on page 6-4 discusses this topic.

WebLogic Server Application Classloader Overview

This section provides an overview of the WebLogic Server application classloaders.

Application Classloading

WebLogic Server classloading is centered on the concept of an application. An application is normally packaged in an Enterprise Archive (EAR) file containing application classes. Everything within an EAR file is considered part of the same application. The following may be part of an EAR or can be loaded as standalone applications:

- An Enterprise JavaBean (EJB) JAR file
- A Web application WAR file
- A resource adapter RAR file

Note: For information on Resource Adapters and classloading, see "About Resource Adapter Classes" on page 6-15.

If you deploy an EJB and a Web application separately, they are considered two applications. If they are deployed together within an EAR file, they are one application. You deploy modules together in an EAR file for them to be considered part of the same application.

Every application receives its own classloader hierarchy; the parent of this hierarchy is the system classpath classloader. This isolates applications so that application A cannot see the classloaders or classes of application B. In hierarchy classloaders, no sibling or friend concepts exist. Application code only has visibility to classes loaded by the classloader associated with the application (or module) and classes that are loaded by classloaders that are ancestors of the application (or module) classloader. This allows WebLogic Server to host multiple isolated applications within the same JVM.

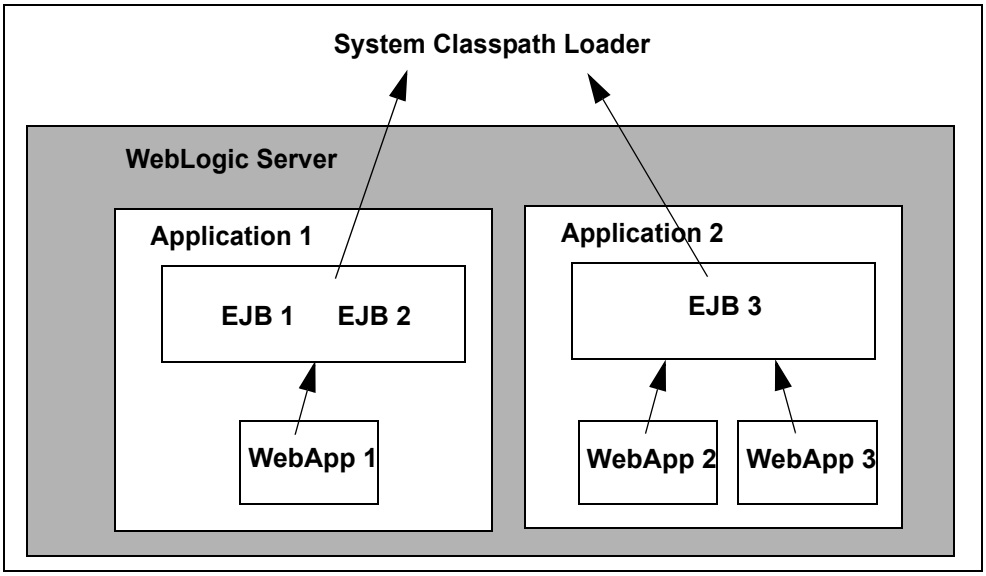
Application Classloader Hierarchy

WebLogic Server automatically creates a hierarchy of classloaders when an application is deployed. The root classloader in this hierarchy loads any EJB JAR files in the application. A child classloader is created for each Web application WAR file.

Because it is common for Web applications to call EJBs, the WebLogic Server application classloader architecture allows JavaServer Page (JSP) files and servlets to see the EJB interfaces in their parent classloader. This architecture also allows Web applications to be redeployed without redeploying the EJB tier. In practice, it is more common to change JSP files and servlets than to change the EJB tier.

The following graphic illustrates this WebLogic Server application classloading concept.

Figure 6-1 WebLogic Server Classloading



If your application includes servlets and JSPs that use EJBs:

- Package the servlets and JSPs in a WAR file
- Package the Enterprise JavaBeans in an EJB JAR file
- Package the WAR and JAR files in an EAR file
- Deploy the EAR file

Although you could deploy the WAR and JAR files separately, deploying them together in an EAR file produces a classloader arrangement that allows the servlets and JSPs to find the EJB classes. If you deploy the WAR and JAR files separately, WebLogic Server creates sibling classloaders for them. This means that you must include the EJB home and remote interfaces in the WAR file, and WebLogic Server must use the RMI stub and skeleton classes for EJB calls, just as it does when EJB clients and implementation classes are in different JVMs. This concept is discussed in more detail in the next section "Application Classloading and Pass-by-Value or Reference" on page 6-14.

Note: The Web application classloader contains all classes for the Web application except for the JSP class. The JSP class obtains its own classloader, which is a child of the Web application classloader. This allows JSPs to be individually reloaded.

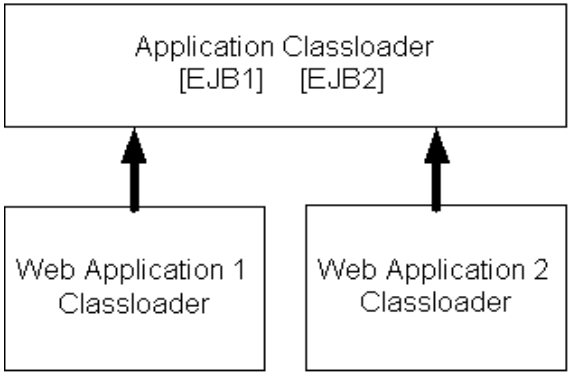
Custom Module Classloader Hierarchies

You can create custom classloader hierarchies for an application allowing for better control over class visibility and reloadability. You achieve this by defining a `classloader-structure` element in the `weblogic-application.xml` deployment descriptor file.

The following diagram illustrates how classloaders are organized by default for WebLogic applications. An application level classloader exists where all EJB classes are loaded. For each Web module, there is a separate child classloader for the classes of that module.

For simplicity, JSP classloaders are not described in the following diagram.

Figure 6-2 Standard Classloader Hierarchy



This hierarchy is optimal for most applications, because it allows call-by-reference semantics when you invoke EJBs. It also allows Web modules to be independently reloaded without affecting other modules. Further, it allows code running in one of the Web modules to load classes from any of the EJB modules. This is convenient, as it can prevent a Web module from including the interfaces for EJBs that is uses. Note that some of those benefits are not strictly J2EE-compliant.

The ability to create custom module classloaders provides a mechanism to declare alternate classloader organizations that allow the following:

- Reloading individual EJB modules independently

- Reloading groups of modules to be reloaded together
- Reversing the parent child relationship between specific Web modules and EJB modules
- Namespace separation between EJB modules

Declaring the Classloader Hierarchy

You can declare the classloader hierarchy in the WebLogic-specific application deployment descriptor `weblogic-application.xml`.

The DTD for this declaration is as follows:

Listing 6-2 Declaring the Classloader Hierarchy

```
<!ELEMENT classloader-structure (module-ref*, classloader-structure*)>

<!ELEMENT module-ref (module-uri)>

<!ELEMENT module-uri (#PCDATA)>
```

The top-level element in `weblogic-application.xml` includes an optional `classloader-structure` element. If you do not specify this element, then the standard classloader is used. Also, if you do not include a particular module in the definition, it is assigned a classloader, as in the standard hierarchy. That is, EJB modules are associated with the application Root classloader, and Web application modules have their own classloaders.

The `classloader-structure` element allows for the nesting of `classloader-structure` stanzas, so that you can describe an arbitrary hierarchy of classloaders. There is currently a limitation of three levels. The outermost entry indicates the application classloader. For any modules not listed, the standard hierarchy is assumed.

Note: JSP classloaders are not included in this definition scheme. JSPs are always loaded into a classloader that is a child of the classloader associated with the Web module to which it belongs.

For more information on the DTD elements, refer to Appendix A, "Enterprise Application Deployment Descriptor Elements."

The following is an example of a classloader declaration (defined in the `classloader-structure` element in `weblogic-application.xml`):

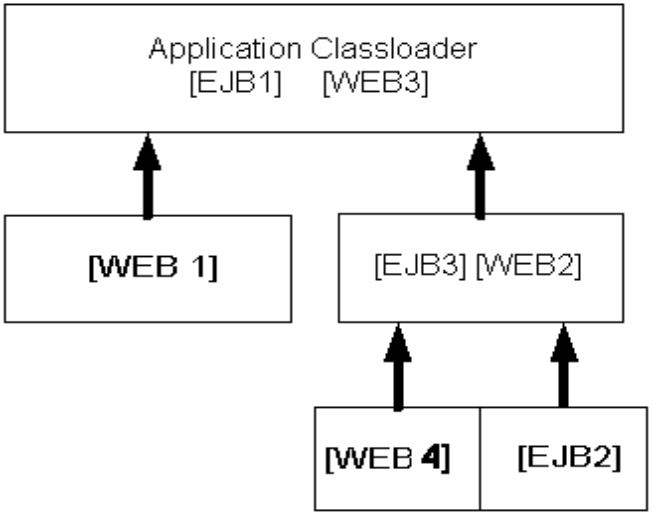
Listing 6-3 Example Classloader Declaration

```
<classloader-structure>
       <module-ref>
              <module-uri>ejb1.jar</module-uri>
       </module-ref>
       <module-ref>
              <module-uri>web3.war</module-uri>
       </module-ref>

       <classloader-structure>
              <module-ref>
                     <module-uri>web1.war</module-uri>
              </module-ref>
       </classloader-structure>

       <classloader-structure>
              <module-ref>
                     <module-uri>ejb3.jar</module-uri>
              </module-ref>
              <module-ref>
                     <module-uri>web2.war</module-uri>
              </module-ref>
              <classloader-structure>
                     <module-ref>
                            <module-uri>web4.war</module-uri>
                     </module-ref>
              </classloader-structure>
```

```
                <classloader-structure>
                        <module-ref>
                                <module-uri>ejb2.jar</module-uri>
                        </module-ref>
                </classloader-structure>
        </classloader-structure>
</classloader-structure>
```

The organization of the nesting indicates the classloader hierarchy. The above stanza leads to a hierarchy shown in the following diagram.

Figure 6-3 Example Classloader Hierarchy



User-Defined Classloader Restrictions

User-defined classloader restrictions give you better control over what is reloadable and provide inter-module class visibility. This feature is primarily for developers. It is useful for iterative development, but the reloading aspect of this feature is not recommended for production use, because it is possible to corrupt a running application if an update includes invalid elements.

Custom classloader arrangements for namespace separation and class visibility are acceptable for production use. However, programmers should be aware that the J2EE specifications say that applications should not depend on any given classloader organization.

Some classloader hierarchies can cause modules within an application to behave more like modules in two separate applications. For example, if you place an EJB in its own classloader so that it can be reloaded individually, you receive call-by-value semantics rather than the call-by-reference optimization BEA provides in our standard classloader hierarchy. Also note that if you use a custom hierarchy, you might end up with stale references. Therefore, if you reload an EJB module, you should also reload calling modules.

There are some restrictions to creating user-defined module classloader hierarchies; these are discussed in the following sections.

Servlet Reloading Disabled

If you use a custom classloader hierarchy, servlet reloading is disabled for Web applications in that particular application.

Nesting Depth

Nesting is limited to three levels (including the application classloader). Deeper nestings lead to a deployment exception.

Module Types

Custom classloader hierarchies are currently restricted to Web and EJB modules.

Duplicate Entries

Duplicate entries lead to a deployment exception.

Interfaces

The standard WebLogic Server classloader hierarchy makes EJB interfaces available to all modules in the application. Thus other modules can invoke an EJB, even though they do not include the interface classes in their own module. This is possible because EJBs are always loaded into the root classloader and all other modules either share that classloader or have a classloader that is a child of that classloader.

With the custom classloader feature, you can configure a classloader hierarchy so that a callee's classes are not visible to the caller. In this case, the calling module must include the interface

classes. This is the same requirement that exists when invoking on modules in a separate application.

Call-by-Value Semantics

The standard classloader hierarchy provided with WebLogic Server allows for calls between modules within an application to use call-by-reference semantics. This is because the caller is always using the same classloader or a child classloader of the callee. With this feature, it is possible to configure the classloader hierarchy so that two modules are in separate branches of the classloader tree. In this case, call-by-value semantics are used.

In-Flight Work

Be aware that the classloader switch required for reloading is not atomic across modules. In fact, updates to applications in general are not atomic. For this reason, it is possible that different in-flight operations (operations that are occurring while a change is being made) might end up accessing different versions of classes depending on timing.

Development Use Only

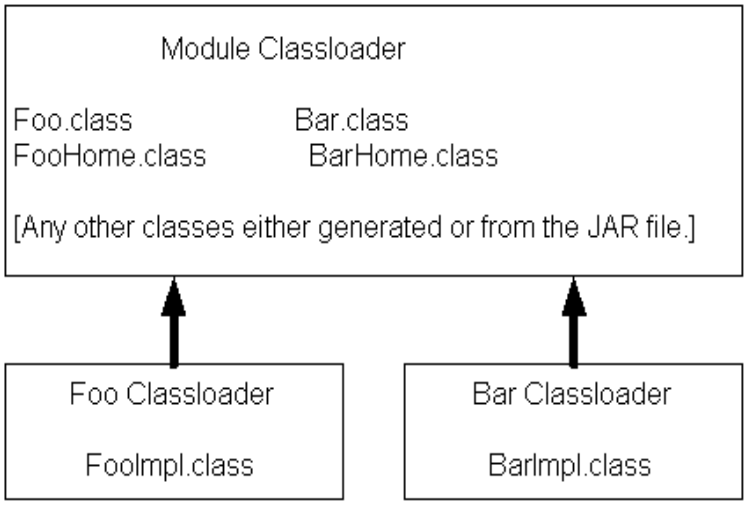
The development-use-only feature is intended for development use. Because updates are not atomic, this feature is not suitable for production use.

Individual EJB Classloader for Implementation Classes

WebLogic Server allows you to reload individual EJB modules without requiring you to reload other modules at the same time and having to redeploy the entire EJB module. This feature is similar to how JSPs are currently reloaded in the WebLogic Server servlet container.

Because EJB classes are invoked through an interface, it is possible to load individual EJB implementation classes in their own classloader. This way, these classes can be reloaded individually without having to redeploy the entire EJB module. Below is a diagram of what the classloader hierarchy for a single EJB module would look like. The module contains two EJBs (`Foo` and `Bar`). This would be a sub-tree of the general application hierarchy described in the previous section.

Figure 6-4 Example Classloader Hierarchy for a Single EJB Module



To perform a partial update of files relative to the root of the exploded application, use the following command line:

Listing 6-4 Performing a Partial File Update

```
java weblogic.Deployer -adminurl url -user user -password password
-name myapp -redeploy myejb/foo.class
```

After the `-redeploy` command, you provide a list of files relative to the root of the exploded application that you want to update. This might be the path to a specific element (as above) or a module (or any set of elements and modules). For example:

Listing 6-5 Providing a List of Relative Files for Update

```
java weblogic.Deployer -adminurl url -user user -password password
-name myapp -redeploy mywar myejb/foo.class anotherejb
```

Given a set of files to be updated, the system tries to figure out the minimum set of things it needs to redeploy. Redeploying only an EJB `impl` class causes only that class to be redeployed. If you specify the whole EJB (in the above example, `anotherejb`) or if you change and update the EJB home interface, the entire EJB module must be redeployed.

Depending on the classloader hierarchy, this redeployment may lead to other modules being redeployed. Specifically, if other modules share the EJB classloader or are loaded into a classloader that is a child to the EJB's classloader (as in the WebLogic Server standard classloader module) then those modules are also reloaded.

Application Classloading and Pass-by-Value or Reference

Modern programming languages use two common parameter passing models: pass-by-value and pass-by-reference. With pass-by-value, parameters and return values are copied for each method call. With pass-by-reference, a pointer (or reference) to the actual object is passed to the method.

Pass by reference improves performance because it avoids copying objects, but it also allows a method to modify the state of a passed parameter.

WebLogic Server includes an optimization to improve the performance of Remote Method Interface (RMI) calls within the server. Rather than using pass by value and the RMI subsystem's marshalling and unmarshalling facilities, the server makes a direct Java method call using pass by reference. This mechanism greatly improves performance and is also used for EJB 2.0 local interfaces.

RMI call optimization and call by reference can only be used when the caller and callee are within the same application. As usual, this is related to classloaders. Because applications have their own classloader hierarchy, any application class has a definition in both classloaders and receives a ClassCastException error if you try to assign between applications. To work around this, WebLogic Server uses call-by-value between applications, even if they are within the same JVM.

Note: Calls between applications are slower than calls within the same application. Deploy modules together as an EAR file to enable fast RMI calls and use of the EJB 2.0 local interfaces.

Resolving Class References Between Modules and Applications

Your applications may use many different Java classes, including Enterprise Beans, servlets and JavaServer Pages, utility classes, and third-party packages. WebLogic Server deploys applications in separate classloaders to maintain independence and to facilitate dynamic redeployment and undeployment. Because of this, you need to package your application classes in such a way that each module has access to the classes it depends on. In some cases, you may have to include a set of classes in more than one application or module. This section describes how WebLogic Server uses multiple classloaders so that you can stage your applications successfully.

About Resource Adapter Classes

With this release of WebLogic Server, each resource adapter now uses its own classloader to load classes (similar to Web applications). As a result, modules like Web applications and EJBs that are packaged along with a resource adapter in an application archive (EAR file) do not have visibility into the resource adapter's classes. If such visibility is required, you must place the resource adapter classes in `APP-INF/classes`. You can also archive these classes (using the JAR utility) and place them in the `APP-INF/lib` of the application archive.

Make sure that no resource-adapter specific classes exist in your WebLogic Server system classpath. If you need to use resource adapter-specific classes with Web modules (for example,

an EJB or Web application), you must bundle these classes in the corresponding module's archive file (for example, the JAR file for EJBs or the WAR file for Web applications).

Packaging Shared Utility Classes

WebLogic Server provides a location within an EAR file where you can store shared utility classes. Place utility JAR files in the `APP-INF/lib` directory and individual classes in the `APP-INF/classes` directory. (Do not place JAR files in the `/classes` directory or classes in the `/lib` directory.) These classes are loaded into the root classloader for the application.

This feature obviates the need to place utility classes in the system classpath or place classes in an EJB JAR file (which depends on the standard WebLogic Server classloader hierarchy). Be aware that using this feature is subtly different from using the manifest `Class-Path` described in the following section. With this feature, class definitions are shared across the application. With manifest `Class-Path`, the classpath of the referencing module is simply extended, which means that separate copies of the classes exist for each module.

Manifest Class-Path

The J2EE specification provides the manifest `Class-Path` entry as a means for a module to specify that it requires an auxiliary JAR of classes. You only need to use this manifest `Class-Path` entry if you have additional supporting JAR files as part of your EJB JAR or WAR file. In such cases, when you create the JAR or WAR file, you must include a manifest file with a `Class-Path` element that references the required JAR files.

The following is a simple manifest file that references a `utility.jar` file:

```
Manifest-Version: 1.0 [CRLF]
Class-Path: utility.jar [CRLF]
```

In the first line of the manifest file, you must always include the `Manifest-Version` attribute, followed by a new line (CR | LF |CRLF) and then the `Class-Path` attribute. More information about the manifest format can be found at:
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR

The manifest `Class-Path` entries refer to other archives relative to the current archive in which these entries are defined. This structure allows multiple WAR files and EJB JAR files to share a common library JAR. For example, if a WAR file contains a manifest entry of `y.jar`, this entry should be next to the WAR file (not within it) as follows:

/<directory>/x.war

/<directory>/y.jars

The manifest file itself should be located in the archive at `META-INF/MANIFEST.MF`.

For more information, see
`http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html`.

Sharing Applications and Modules By Using J2EE Libraries

This release of WebLogic Server includes a new feature, J2EE libraries, that provides an easy way to share one or more different types of J2EE modules among multiple Enterprise Applications. A J2EE library is a single module or collection of modules that is registered with the J2EE application container upon deployment. For more information, see Chapter 8, "Creating Shared J2EE Libraries and Optional Packages."

Adding JARs to the System Classpath

WebLogic Server 9.0 introduces a `lib` subdirectory, located in the domain directory, that you can use to add one or more JAR files to the WebLogic Server system classpath when servers start up. The `lib` subdirectory is intended for JAR files that change infrequently and are required by all or most applications deployed in the server, or by WebLogic Server itself. For example, you might use the `lib` directory to store third-party utility classes that are required by all deployments in a domain. You can also use it to apply patches to WebLogic Server.

The `lib` directory is not recommended as a general-purpose method for sharing a JARs between one or two applications deployed in a domain, or for sharing JARs that need to be updated periodically. If you update a JAR in the `lib` directory, you must reboot all servers in the domain in order for applications to realize the change. If you need to share a JAR file or J2EE modules among several applications, use the J2EE libraries feature described in "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.

To share JARs using the `lib` directory:

1. Shutdown all servers in the domain.
2. Copy the JAR file(s) to share into a `lib` subdirectory of the domain directory. For example:

```
mkdir c:\bea\weblogic90\samples\domains\wl_server\lib
cp c:\3rdpartyjars\utility.jar
   c:\bea\weblogic90\samples\domains\wl_server\lib
```

Note: WebLogic Server must have read access to the `lib` directory during startup.

Note: The Administration Server does not automatically copy files in the `lib` directory to Managed Servers on remote machines. If you have Managed Servers that do not share the same physical domain directory as the Administration Server, you must manually

copy JAR file(s) to the *domain_name*/lib directory on the Managed Server machines.

3. Start the Administration Server and all Managed Servers in the domain. WebLogic Server appends JAR files found in the lib directory to the system classpath. Multiple files are added in alphabetical order.

CHAPTER 7

Developing Applications for Production Redeployment

The following sections describes how to program and maintain applications use the production redeployment strategy:

What is Production Redeployment?

Production redeployment enables an Administrator to redeploy a new version of an application in a production environment without stopping the deployed application or otherwise interrupting the application's availability to clients. Production redeployment works by deploying a new version of an updated application alongside an older version of the same application. WebLogic Server automatically manages client connections so that only new client requests are directed to the new version. Clients already connected to the application during the redeployment continue to use the older, retiring version of the application until they complete their work.

See Using Production Redeployment to Upgrade Applications for more information.

Supported and Unsupported Application Types

Production redeployment is supported primarily for applications with a Web application entry point (HTTP clients). WebLogic Server 9.0 can automatically manage HTTP client entry points to isolate connections to the newer and older application versions. This means that production redeployment is supported for standalone Web Application modules, and for Enterprise Applications that are accessed via an embedded Web Application module.

Applications that are accessed by Java clients, including applets, are specifically not supported with production redeployment. Java clients that attempt a JNDI lookup of global bindings for a versioned application receive a warning. These types of lookups must be avoided because they interfere with WebLogic Server's automatic management of client entry points during application retirement and can cause an application version to be retired prematurely. Clients can disable this checking by setting `weblogic.jndi.WLContext.ALLOW_EXTERNAL_APP_LOOKUP` to `true` when performing JNDI lookups.

Enterprise Applications can contain any of the supported J2EE module types except Web Services modules. Web Services modules are not supported for production redeployment, even if you package the service in a WAR file. When a production redeployment operation is requested, the WebLogic Server deployment API checks for the presence of Web Services modules and throws an exception if one is found. Enterprise Applications can also include application-scoped JMS and JDBC modules.

If an Enterprise Application includes a JCA resource adapter module, the module:

- Must be JCA 1.5 compliant
- Must implement the `weblogic.connector.extensions.Suspendable` interface

- Must be used in an application-scoped manner, having `enable-access-outside-app` set to "true" (the default value).

Before resource adapters in a newer version of the EAR are deployed, resource adapters in the older application version receive a callback. WebLogic Server then deploys the newer application version and retires the entire older version of the EAR.

Additional Application Support

Additional production redeployment support is provided for Enterprise Applications that are accessed by inbound JMS messages from a global JMS destination, and that use one or more message-driven beans as consumers. For this type of application, WebLogic Server suspends message-driven beans in the older, retiring application version before deploying message-driven beans in the newer version. Production redeployment is not supported with JMS consumers that use the JMS API for global JMS destinations. If the message-driven beans need to receive all messages published from topics, including messages published while bean are suspended, use durable subscribers.

Programming Requirements and Conventions

WebLogic Server performs production redeployment by deploying two instances of an application simultaneously. You must observe certain programming conventions to ensure that multiple instances of the application can co-exist in a WebLogic Server domain. The following sections describe each programming convention required for using production redeployment.

Applications Should Be Self-Contained

As a best practice, applications that use the in-place redeployment strategy should be self-contained in their use of resources. This means you should generally use application-scoped JMS and JDBC resources, rather than global resources, whenever possible for versioned applications.

If an application must use a global resource, you must ensure that the application supports safe, concurrent access by multiple instances of the application. This same restriction also applies if the application uses external (separately-deployed) applications, or uses an external property file. WebLogic Server does not prevent the use of global resources with versioned applications, but you must ensure that resources are accessed in a safe manner.

Looking up a global JNDI resource from within a versioned application results in a warning message. To disable this check, set the JNDI environment property

`weblogic.jndi.WLContext.ALLOW_GLOBAL_RESOURCE_LOOKUP` to `true` when performing the JNDI lookup.

Similarly, looking up an external application results in a warning unless you set the JNDI environment property, `weblogic.jndi.WLContext.ALLOW_EXTERNAL_APP_LOOKUP`, to `true`.

Versioned Applications Access the Current Version JNDI Tree by Default

WebLogic Server binds application-scoped resources, such as JMS and JDBC application modules, into a local JNDI tree available to the application. As with non-versioned applications, versioned applications can look up application-scoped resources directly from this local tree. Application-scoped JMS modules can be accessed via any supported JMS interfaces, such as the JMS API or a message-driven bean.

Application modules that are bound to the global JNDI tree should be accessed only from within the same application version. WebLogic Server performs version-aware JNDI lookups and bindings for global resources deployed in a versioned application. By default, an internal JNDI lookup of a global resource returns bindings for the same version of the application.

If the current version of the application cannot be found, you can use the JNDI environment property `weblogic.jndi.WLContext.RELAX_VERSION_LOOKUP` to return bindings from the currently active version of the application, rather than the same version.

Warning: Set `weblogic.jndi.WLContext.RELAX_VERSION_LOOKUP` to `true` only if you are certain that the newer and older version of the resource that you are looking up are compatible with one another.

Security Providers Must Be Compatible

Any security provider used in the application must support the WebLogic Server application versioning SSPI. The default WebLogic Server security providers for authorization, role mapping, and credential mapping support the application versioning SSPI.

Applications Must Specify a Version Identifier

In order to use production redeployment, both the current, deployed version of the application and the updated version of the application must specify unique version identifiers. See "Assigning an Application Version" on page 7-5.

Applications Can Access Name and Identifier

Versioned applications can programmatically obtain both an application name, which remains constant across different versions, and an application identifier, which changes to provide a unique label for different versions of the application. Use the application name for basic display or error messages that refer to the application's name irrespective of the deployed version. Use the application ID when the application must provide unique identifier for the deployed version of the application. See "Accessing Version Information" on page 7-7 for more information about the MBean attributes that provide the name and identifier.

Client Applications Use Same Version when Possible

As described in "What is Production Redeployment?" on page 7-2, WebLogic Server attempts to route a client application's requests to the same version of the application until all of the client's in-progress work has completed. However, if an application version is retired using a timeout period, or is undeployed, the client's request will be routed to the active version of the application. In other words, a client's association with a given version of an application is maintained only on a "best-effort basis."

This behavior can be problematic for client applications that recursively access other applications when processing requests. WebLogic Server attempts to dispatch requests to the same versions of the recursively-accessed applications, but cannot guarantee that an intermediate application version is not undeployed manually or after a timeout period. If you have a group of related applications with strict version requirements, BEA recommends packaging all of the applications together to ensure version consistency during production redeployment.

Assigning an Application Version

BEA recommends that you specify the version identifier in the `MANIFEST.MF` of the application, and automatically increment the version each time a new application is released for deployment. This ensures that production redeployment is always performed when the administrator or deployer redeploys the application.

For testing purposes, a deployer can also assign a version identifier to an application during deployment and redeployment. See Assigning a Version Identifier During Deployment and Redeployment in *Deploying Applications to WebLogic Server*.

Application Version Conventions

WebLogic Server obtains the application version from the value of the `Weblogic-Application-Version` property in the `MANIFEST.MF` file. The version string can be a maximum of 215 characters long, and must consist of valid characters as identified in Figure 7-1.

Table 7-1 Valid and Invalid Characters

Valid ASCII Characters	Invalid Version Constructs
a-z	..
A-Z	.
0-9	
period (“.”), underscore (“_”), or hyphen (“-”) in combination with other characters	

For example, the following manifest file content describes an application with version “`v1`”:

```
Manifest-Version: 1.0
     Created-By: 1.4.1_05-b01 (Sun Microsystems Inc.)
     Weblogic-Application-Version: v1
```

Upgrading Applications to Use Production Redeployment

If you are upgrading applications for deployment to WebLogic Server 9.0, note that the `Name` attribute retrieved from `AppDeploymentMBean` now returns a unique application identifier consisting of both the deployed application name and the application version string. Applications that require only the deployed application name must use the new `ApplicationName` attribute instead of the `Name` attribute. Applications that require a unique identifier can use either the `Name` or `ApplicationIdentifier` attribute, as described in “Accessing Version Information” on page 7-7.

Accessing Version Information

Your application code can use new MBean attributes to retrieve version information for display, logging, or other uses. Figure 7-2 describes the read-only attributes provided by `ApplicationMBean`.

Table 7-2 Read-Only Version Attributes in ApplicationMBean

Attribute Name	Description
`ApplicationName`	A String that represents the deployment name of the application
`VersionIdentifier`	A String that uniquely identifies the current application version across all versions of the same application
`ApplicationIdentifier`	A String that uniquely identifies the current application version across all deployed applications and versions

`ApplicationRuntimeMBean` also provides version information in the new read-only attributes described in Figure 7-3, "Read-Only Version Attributes in ApplicationRuntimeMBean," on page 7-7.

Table 7-3 Read-Only Version Attributes in ApplicationRuntimeMBean

Attribute Name	Description
`ApplicationName`	A String that represents the deployment name of the application

Attribute Name	Description
`ApplicationVersion`	A string that represents the version of the application.
`ActiveVersionState`	An integer that indicates the current state of the active application version. Valid states for an active version are: • ACTIVATED—indicates that one or more modules of the application are active and available for processing new client requests. • PREPARED—indicates that WebLogic Server has prepared one or more modules of the application, but that it is not yet active. • UNPREPARED—indicates that no modules of the application are prepared or active. See the WebLogic Server 9.0 API Reference for more information. Note that the currently active version does not always correspond to the last-deployed version, because the Administrator can reverse the production redeployment process. See Rolling Back the Production Redeployment Process in *Deploying Applications to WebLogic Server*.

CHAPTER 8

Creating Shared J2EE Libraries and Optional Packages

The following sections describe how to share components and classes among applications using shared J2EE libraries and optional packages:

Overview of Shared J2EE Libraries and Optional Packages

Prior to WebLogic Server 9.0, multiple Enterprise Applications could not easily share a single J2EE module or a collection of modules. Sharing J2EE modules required you to either package a copy of the modules in multiple EARs, or add the paths to the shared modules to the system classpath and add duplicate deployment descriptors for the shared modules into each application that referenced them. Copying modules made subsequent application updates difficult, because an update to a shared module required re-copying and re-packaging all Enterprise Applications that used the module. Adding modules to the system classpath also made updates difficult, because it required rebooting the WebLogic Server instance in order to use an updated module.

The shared J2EE library feature in WebLogic Server 9.0 provides an easy way to share one or more different types of J2EE modules among multiple Enterprise Applications. A shared J2EE library is a single module or collection of modules that is registered with the J2EE application container upon deployment. A shared J2EE library can be any of the following:

- standalone EJB module
- standalone Web application module
- multiple EJB modules packaged in an Enterprise Application
- multiple Web application modules package in an Enterprise Application
- single plain JAR file

BEA recommends that you package a shared J2EE library into its appropriate archive file (EAR, JAR, or WAR). However, for development purposes, you may choose to deploy shared J2EE libraries as exploded archive directories to facilitate repeated updates and redeployments.

After the shared J2EE library has been registered, you can deploy Enterprise Applications that reference the library. Each referencing application receives a reference to the required library on deployment, and can use the modules that make up the library as if they were packaged as part of the referencing application itself. The library classes are added to the classpath of the referencing application, and the referencing application's deployment descriptors are merged (in memory) with those of the modules that make up the shared J2EE library.

In general, this topic discusses shared J2EE libraries that can be referenced only by Enterprise Applications. You can also create libraries that can be referenced only by another Web application. The functionality is very similar to application libraries, although the method of referencing them is slightly different. See "Web Application Shared J2EE Library Information" on page 8-19 for details.

Note: WebLogic Server 9.0 also provides a simple way to add one or more JAR files to the WebLogic Server System classpath, using the `lib` subdirectory of the domain directory. See "Adding JARs to the System Classpath" on page 6-17.

Optional Packages

WebLogic Server supports optional packages as described in the J2EE 1.4 Specification, Section 8.2 Optional Package Support, with versioning described in Optional Package Versioning. Optional packages provide similar functionality to J2EE libraries, allowing you to easily share a single JAR file among multiple applications. As with J2EE libraries, optional packages must first be registered with WebLogic Server by deploy the associated JAR file as an optional package. After registering the package, you can deploy J2EE modules that reference the package in their manifest files.

Optional packages differ from J2EE libraries because optional packages can be referenced from any J2EE module (EAR, JAR, WAR, or RAR archive) or exploded archive directory. J2EE libraries can be referenced only from a valid Enterprise Application.

For example, third-party Web Application Framework classes needed by multiple Web Applications can be packaged and deployed in a single JAR file, and referenced by multiple Web Application modules in the domain. Optional packages, rather than J2EE libraries, are used in this case, because the individual Web Application modules must reference the shared JAR file. (With J2EE libraries, only a complete Enterprise Application can reference the library).

Note: BEA documentation and WebLogic Server utilities use the term *library* to refer to both J2EE libraries and optional packages. Optional packages are called out only when necessary.

Versioning Support for Libraries

WebLogic Server supports versioning of shared J2EE libraries, so that referencing applications can specify a required minimum version of the library to use, or an exact, required version. WebLogic Server supports two levels of versioning for shared J2EE libraries, as described in the Optional Package Versioning document:

- Specification Version—Identifies the version number of the specification (for example, the J2EE specification version) to which a shared J2EE library or optional package conforms.
- Implementation Version—Identifies the version number of the actual code implementation for the library or package. For example, this would correspond to the actual revision number or release number of your code. Note that you must also provide a specification version in order to specify an implementation version.

As a best practice, BEA recommends that you always include version information (an implementation version, or both an implementation and specification version) when creating shared J2EE libraries. Creating and updating version information as you develop shared components allows you to deploy multiple versions of those components simultaneously for testing. If you include no version information, or fail to increment the version string, then you must undeploy existing libraries before you can deploy the newer one. See "Deploying Shared J2EE Libraries and Dependent Applications" on page 8-18.

Versioning information in the referencing application determines the library and package version requirements for that application. Different applications can require different versions of a given library or package. For example, a production application may require a specific version of a library, because only that library has been fully approved for production use. An internal application may be configured to always use a minimum version of the same library. Applications that require no specific version can be configured to use the latest version of the library. "Referencing Shared J2EE Libraries in an Enterprise Application" on page 8-11.

Shared J2EE Libraries and Optional Packages Compared

Optional packages and shared J2EE libraries have the following features in common:

- Both are registered with WebLogic Server instances at deployment time.
- Both support an optional implementation version and specification version string.
- Applications that reference shared J2EE libraries and optional packages can specify required versions for the shared files.

Optional packages differ from shared J2EE Libraries in the following basic ways:

- Optional packages are plain JAR files, whereas shared J2EE libraries can be plain JAR files, J2EE Enterprise Applications, or standalone J2EE modules (EJB and Web applications). This means that libraries can have valid J2EE and WebLogic Server deployment descriptors. Any deployment descriptors in an optional package JAR file are ignored.
- Any J2EE application or module can reference an optional package (using `META-INF/MANIFEST.MF`), whereas only Enterprise Applications and Web applications can reference a shared J2EE library (using `weblogic-application.xml` or `weblogic.xml`)
- Optional packages can reference other optional packages, but shared J2EE libraries cannot reference other shared J2EE libraries.

In general, use shared J2EE libraries when you need to share one or more EJB, Web Application or Enterprise Application modules among different Enterprise Applications. Use optional packages when you need to share one or more classes (packaged in a JAR file) among different J2EE modules.

Plain JAR files can be shared either as libraries or optional packages. Use optional packages if you want to:

- Share a plain JAR file among multiple J2EE modules
- Reference shared JAR files from other shared JARs
- Share plain JARs as described by the J2EE 1.4 specification

Use shared J2EE libraries to share a plain JAR file if you only need to reference the JAR file from one or more Enterprise Applications, and you do not need to maintain strict compliance with the J2EE specification.

Note: BEA documentation and WebLogic Server utilities use the term *shared J2EE library* to refer to both libraries and optional packages. Optional packages are called out only when necessary.

Additional Information

For information about deploying and managing shared J2EE libraries, optional packages, and referencing applications from the Administrator's perspective, see Deploying Shared J2EE Libraries and Dependent Applications in *Deploying Applications to WebLogic Server*.

Creating Shared J2EE Libraries

To create a new shared J2EE library that you can share with multiple applications:

1. Assemble the shared J2EE library into a valid, deployable J2EE module or Enterprise Application. The library must have the required J2EE deployment descriptors for the J2EE module or for an Enterprise Application.

   See "Assembling Shared J2EE Library Files" on page 8-6.

2. Assemble optional package classes into a working directory.

   See "Assembling Optional Package Class Files" on page 8-7.

3. Create and edit the `MANIFEST.MF` file for the shared J2EE library to specify the name and version string information.

See "Editing Manifest Attributes for Shared J2EE Libraries" on page 8-7.

4. Package the shared J2EE library for distribution and deployment.

   See "Packaging Shared J2EE Libraries for Distribution and Deployment" on page 8-10.

Assembling Shared J2EE Library Files

The following types of J2EE modules can be deployed as a shared J2EE library:

- An EJB module, either an exploded directory or packaged in a JAR file.
- A Web Application module, either an exploded directory or packaged in a WAR file.
- An Enterprise application, either an exploded directory or packaged in an EAR file.
- A plain Java class or classes packaged in a JAR file.

Shared J2EE libraries have the following restrictions:

- You cannot reference a shared J2EE library from another library.
- You must ensure that context roots in Web application modules of the shared J2EE library do not conflict with context roots in the referencing Enterprise Application. If necessary, you can configure referencing applications to override a library's context root. See "Referencing Shared J2EE Libraries in an Enterprise Application" on page 8-11.
- Shared J2EE libraries cannot be nested. For example, if you are deploying an EAR as a shared J2EE library, the entire EAR must be designated as the library. You cannot designate individual J2EE modules within the EAR as separate, named libraries.
- As with any other J2EE module or Enterprise Application, a shared J2EE library must be configured for deployment to the target servers or clusters in your domain. This means that a library requires valid J2EE deployment descriptors as well as WebLogic Server-specific deployment descriptors and an optional deployment plan. See *Deploying Applications to WebLogic Server*.

BEA recommends packaging shared J2EE libraries as Enterprise Applications, rather than as standalone J2EE modules. This is because the URI of a standalone module is derived from the deployment name, which can change depending on how the module is deployed. By default, WebLogic Server uses the deployment archive filename or exploded archive directory name as the deployment name. If you redeploy a standalone shared J2EE library from a different file or location, the deployment name and URI also change, and referencing applications that use the wrong URI cannot access the deployed library.

If you choose to deploy a shared J2EE library as a standalone J2EE module, always specify a known deployment name during deployment and use that name as the URI in referencing applications.

Assembling Optional Package Class Files

Any set of classes can be organized into an optional package file. The collection of shared classes will eventually be packaged into a standard JAR archive. However, because you will need to edit the manifest file for the JAR, begin by assembling all class files into a working directory:

1. Create a working directory for the new optional package. For example:

   ```
   mkdir /apps/myOptPkg
   ```

2. Copy the compiled class files into the working directory, creating the appropriate package sudirectories as necessary. For example:

   ```
   mkdir -p /apps/myOptPkg/org/myorg/myProduct
   cp /build/classes/myOptPkg/org/myOrg/myProduct/*.class
   /apps/myOptPkg/org/myOrg/myProduct
   ```

3. If you already have a JAR file that you want to use as an optional package, extract its contents into the working directory so that you can edit the manifest file:

   ```
   cd /apps/myOptPkg
   jar xvf /build/libraries/myLib.jar
   ```

Editing Manifest Attributes for Shared J2EE Libraries

The name and version information for a shared J2EE library are specified in the `META-INF/MANIFEST.MF` file. Table 8-1describes the valid shared J2EE library manifest attributes.

Table 8-1 Manifest Attributes for J2EE Libraries

Attribute	Description
`Extension-Name`	An optional string value that identifies the name of the shared J2EE library. Referencing applications must use the exact `Extension-Name` value to use the library. As a best practice, always specify an `Extension-Name` value for each library. If you do not specify an extension name, one is derived from the deployment name of the library. Default deployment names are different for archive and exploded archive deployments, and they can be set to arbitrary values in the deployment command.

Attribute	Description
`Specification-Version`	An optional String value that defines the specification version of the shared J2EE library. Referencing applications can optionally specify a required `Specification-Version` for a library; if the exact specification version is not available, deployment of the referencing application fails. The `Specification-Version` uses the following format: Major/minor version format, with version and revision numbers separated by periods (such as "9.0.1.1") Referencing applications can be configured to require either an exact version of the shared J2EE library, a minimum version, or the latest available version. The specification version for a shared J2EE library can also be set at the command-line when deploying the library, with some restrictions. See "Deploying Shared J2EE Libraries and Dependent Applications" on page 8-18.
`Implementation-Version`	An optional String value that defines the code implementation version of the shared J2EE library. You can provide an `Implementation-Version` only if you have also defined a `Specification-Version`. `Implementation-Version` uses the following formats: • Major/minor version format, with version and revision numbers separated by periods (such as "9.0.1.1") • Text format, with named versions (such as "9011Beta" or "9.0.1.1.B") If you use the major/minor version format, referencing applications can be configured to require either an exact version of the shared J2EE library, a minimum version, or the latest available version. If you use the text format, referencing applications must specify the exact version of the library. The implementation version for a shared J2EE library can also be set at the command-line when deploying the library, with some restrictions. See "Deploying Shared J2EE Libraries and Dependent Applications" on page 8-18.

To specify attributes in a manifest file:

1. Open (or create) the manifest file using a text editor. For the example shared J2EE library, you would use the commands:

   ```
   cd /apps/myLibrary
   mkdir META-INF
   emacs META-INF/MANIFEST.MF
   ```

   For the optional package example, use:

   ```
   cd /apps/myOptPkg
   mkdir META-INF
   emacs META-INF/MANIFEST.MF
   ```

2. In the text editor, add a string value to specify the name of the shared J2EE library. For example:

   ```
   Extension-Name: myExtension
   ```

   Applications that reference the library must specify the exact `Extension-Name` in order to use the shared files.

3. As a best practice, enter the optional version information for the shared J2EE library. For example:

   ```
   Extension-Name: myExtension
   Specification-Version: 2.0
   Implementation-Version: 9.0.0
   ```

   Using the major/minor format for the version identifiers provides the most flexibility when referencing the library from another application (see Table 8-1 on page 8)

   Note: Although you can optionally specify the `Specification-Version` and `Implementation-Version` at the command-line during deployment, BEA recommends that you include these strings in the `MANIFEST.MF` file. Including version strings in the manifest ensures that you can deploy new versions of the library alongside older versions. See "Deploying Shared J2EE Libraries and Dependent Applications" on page 8-18.

Packaging Shared J2EE Libraries for Distribution and Deployment

If you are delivering the shared J2EE Library or optional package for deployment by an Administrator, package the deployment files into an archive file (an `.EAR` file or standalone module archive file for shared J2EE libraries, or a simple `.JAR` file for optional packages) for distribution. See "Deploying and Packaging from a Split Development Directory" on page 5-1.

Because a shared J2EE library is packaged as a standard J2EE application or standalone module, you may also choose to export a library's deployment configuration to a deployment plan, as described in *Deploying Applications to WebLogic Server*. Optional package `.JAR` files contain no deployment descriptors and cannot be exported.

For development purposes, you may choose to deploy libraries as exploded archive directories to facilitate repeated updates and redeployments.

Referencing Shared J2EE Libraries in an Enterprise Application

A J2EE application can reference a registered shared J2EE library using entries in the application's `weblogic-application.xml` deployment descriptor. Table 8-2 describes the XML elements that define a library reference.

Table 8-2 weblogic-application.xml Elements for Referencing a Shared J2EE Library

Element	Description
`library-ref`	`library-ref` is the parent element in which you define a reference to a shared J2EE library. Enclose all other elements within `library-ref`.
`library-name`	A required string value that specifies the name of the shared J2EE library to use. `library-name` must exactly match the value of the `Extension-Name` attribute in the library's manifest file. (See Table 8-1.)
`specification-version`	An optional String value that defines the required specification version of the shared J2EE library. If this element is not set, the application uses a matching library with the highest specification version. If you specify a string value using major/minor version format, the application uses a matching library with the highest specification version that is not below the configured value. If all available libraries are below the configured `specification-version`, the application cannot be deployed. The required version can be further constrained by using the `exact-match` element, described below. If you specify a String value that does not use major/minor versioning conventions (for example, 9.0BETA) the application requires a shared J2EE library having the exact same string value in the `Specification-Version` attribute in the library's manifest file. (See Table 8-1 on page 8.)

Element	Description
`implementation-version`	An optional String value that specifies the required implementation version of the shared J2EE library. If this element is not set, the application uses a matching library with the highest implementation version. If you specify a string value using major/minor version format, the application uses a matching library with the highest implementation version that is not below the configured value. If all available libraries are below the configured `implementation-version`, the application cannot be deployed. The required implementation version can be further constrained by using the `exact-match` element, described below. If you specify a String value that does not use major/minor versioning conventions (for example, 9.0BETA) the application requires a shared J2EE library having the exact same string value in the `Implementation-Version` attribute in the library's manifest file. (See Table 8-1 on page 8.)
`exact-match`	An optional boolean value that determines whether the application should use a shared J2EE library with a higher specification or implementation version than the configured value, if one is available. By default this element is false, which means that WebLogic Server uses higher-versioned libraries if they are available. Set this element to true to require the exact matching version as specified in the `specification-version` and `implementation-version` elements.
`context-root`	An optional String value that provides an alternate context root to use for a Web application shared J2EE library. Use this element if the context root of a library conflicts with the context root of a Web Application in the referencing J2EE application. *Web application shared J2EE library* refers to special kind of library: a Web application that is referenced by another Web application. See "Web Application Shared J2EE Library Information" on page 8-19.

For example, this simple entry in the `weblogic-application.xml` descriptor references a shared J2EE library, `myLibrary`:

```
<library-ref>
   <library-name>myLibrary</library-name>
</library-ref>
```

In the above example, WebLogic Server attempts to find a library name `myLibrary` when deploying the dependent application. If more than one copy of `myLibrary` is registered,

WebLogic Server selects the library with the highest specification version. If multiple copies of the library use the selected specification version, WebLogic Server selects the copy having the highest implementation version.

This example references a shared J2EE library with a requirement for the specification version:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
</library-ref>
```

In the above example, WebLogic Server looks for matching libraries having a specification version of 2.0 or higher. If multiple libraries are at or above version 2.0, WebLogic Server examines the selected libraries that use Float values for their implementation version and selects the one with the highest version. Note that WebLogic Server ignores any selected libraries that have a non-Float value for the implementation version.

This example references a shared J2EE library with both a specification version and a non-Float value implementation version:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
   <implementation-version>81Beta</implementation-version>
</library-ref>
```

In the above example, WebLogic Server searches for a library having a specification version of 2.0 or higher, and having an exact match of `81Beta` for the implementation version.

The following example requires an exact match for both the specification and implementation versions:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
   <implementation-version>8.1</implementation-version>
   <exact-match>true</exact-match>
</library-ref>
```

URIs for Shared J2EE Libraries Deployed As a Standalone Module

When referencing the URI of a shared J2EE library that was deployed as a standalone module (EJB or Web Application), note that the module URI corresponds to the deployment name of the shared J2EE library. This can be a name that was manually assigned during deployment, the name of the archive file that was deployed, or the name of the exploded archive directory that was deployed. If you redeploy the same module using a different file name or from a different location, the default deployment name also changes and referencing applications must be updated to use the correct URI.

To avoid this problem, deploy all shared J2EE libraries as Enterprise Applications, rather than as standalone modules. If you choose to deploy a library as a standalone J2EE module, always specify a known deployment name and use that name as the URI in referencing applications.

Referencing Optional Packages from a J2EE Application or Module

Any J2EE archive (JAR, WAR, RAR, EAR) can reference one or more registered optional packages using attributes in the archive's manifest file.

Table 8-3 Manifest Attributes for Referencing Optional Packages

Attribute	Description
`Extension-List` *`logical_name`* `[...]`	A required String value that defines a logical name for an optional package dependency. You can use multiple values in the `Extension-List` attribute to designate multiple optional package dependencies. For example: `Extension-List: dependency1 dependency2`
`[`*`logical_name`*`-]Extension-Name`	A required string value that identifies the name of an optional package dependency. This value must match the `Extension-Name` attribute defined in the optional package's manifest file. If you are referencing multiple optional packages from a single archive, prepend the appropriate logical name to the `Extension-Name` attribute. For example: `dependency1-Extension-Name: myOptPkg`

Attribute	Description
[*logical_name*-]Specification-Version	An optional String value that defines the required specification version of an optional package. If this element is not set, the archive uses a matching package with the highest specification version. If you include a `specification-version` value using the major/minor version format, the archive uses a matching package with the highest specification version that is not below the configured value. If all available package are below the configured `specification-version`, the archive cannot be deployed. If you specify a String value that does not use major/minor versioning conventions (for example, 9.0BETA) the archive requires a matching optional package having the exact same string value in the `Specification-Version` attribute in the package's manifest file. (See Table 8-1 on page 8.) If you are referencing multiple optional packages from a single archive, prepend the appropriate logical name to the `Specification-Version` attribute.
[*logical_name*-]Implementation-Version	An optional String value that specifies the required implementation version of an optional package. If this element is not set, the archive uses a matching package with the highest implementation version. If you specify a string value using the major/minor version format, the archive uses a matching package with the highest implementation version that is not below the configured value. If all available libraries are below the configured `implementation-version`, the application cannot be deployed. If you specify a String value that does not use major/minor versioning conventions (for example, 9.0BETA) the archive requires a matching optional package having the exact same string value in the `Implementation-Version` attribute in the package's manifest file. (See Table 8-1 on page 8.) If you are referencing multiple optional packages from a single archive, prepend the appropriate logical name to the `Implementation-Version` attribute.

For example, this simple entry in the manifest file for a dependent archive references two optional packages, `myAppPkg` and `my3rdPartyPkg`:

```
Extension-List: internal 3rdparty
internal-Extension-Name: myAppPkg
3rdparty-Extension-Name: my3rdPartyPkg
```

This example requires a specification version of 2.0 or higher for `myAppPkg`:

```
Extension-List: internal 3rdparty
internal-Extension-Name: myAppPkg
3rdparty-Extension-Name: my3rdPartyPkg
internal-Specification-Version: 2.0
```

This example requires a specification version of 2.0 or higher for `myAppPkg`, and an exact match for the implementation version of `my3rdPartyPkg`:

```
Extension-List: internal 3rdparty
internal-Extension-Name: myAppPkg
3rdparty-Extension-Name: my3rdPartyPkg
internal-Specification-Version: 2.0
3rdparty-Implementation-Version: 8.1GA
```

By default, when WebLogic Server deploys an application or module and it cannot resolve a reference in the application's manifest file to an optional package, WebLogic Server prints a warning, but continues with the deployment anyway. You can change this behavior by setting the system property `weblogic.application.RequireOptionalPackages` to `true` when you start WebLogic Server, either at the command line or in the command script file from which you start the server. Setting this system property to `true` means that WebLogic Server does *not* attempt to deploy an application or module if it cannot resolve an optional package reference in its manifest file.

Using weblogic.appmerge to Merge Libraries

`weblogic.appmerge` is a tool that is used to merge libraries into an application, with merged contents and merged descriptors. It also has the ability to write a merged application to disk. You can then use `weblogic.appmerge` to understand a library merge by examining the merged application you have written to disk.

- "Using weblogic.appmerge from the CLI" on page 8-17
- "Using weblogic.appmerge as an Ant Task" on page 8-17

Using weblogic.appmerge from the CLI

Invoke weblogic.appmerge using the following syntax:

```
java weblogic.appmerge [options] <ear, jar, war file, or directory>
```

where valid options are shown in Table 8-4:

Table 8-4 weblogic.appmerge Options

Option	Comment
`-help`	Print the standard usage message.
`-version`	Print version information.
`-output <file>`	Specifies an alternate output archive or directory. If not set, output is placed in the source archive or directory.
`-plan <file>`	Specifies an optional deployment plan.
-verbose	Provide more verbose output.
`-library <file>`	Comma-separated list of libraries. Each library may optionally set its name and versions, if not already set in its manifest, using the following syntax: `<file> [@name=<string>@libspecver=<version> @libimplver=<version|string>].`
`-librarydir <dir>`	Registers all files in specified directory as libraries.

Example:

```
$ java weblogic.appmerge -output CompleteSportsApp.ear -library Weather
.war,Calendar.ear SportsApp.ear
```

Using weblogic.appmerge as an Ant Task

The ant task provides similar functionality as the command line utility. It supports `source`, `output`, `libraryDir`, `plan` and `verbose` attributes as well as multiple `<library>` sub-elements. Here is an example:

```
<taskdef name="appmerge" classname="weblogic.ant.taskdefs.j2ee.AppMergeTas
k"/>
```

```
<appmerge source="SportsApp.ear" output="CompleteSportsApp.ear">
   <library file="Weather.war"/>
   <library file="Calendar.ear"/>
</appmerge>
```

Integrating Shared J2EE Libraries with the Split Development Directory Environment

The `BuildXMLGen` includes a `-librarydir` option to generate build targets that include one or more shared J2EE library directories. See "Generating a Basic build.xml File Using weblogic.BuildXMLGen" on page 3-13.

The `wlcompile` and `wlappc` Ant tasks include a `librarydir` attribute and `library` element to specify one or more shared J2EE library directories to include in the classpath for application builds. See "Building Applications in a Split Development Directory" on page 4-1.

Deploying Shared J2EE Libraries and Dependent Applications

Shared J2EE libraries are registered with one or more WebLogic Server instances by deploying them to the target servers and indicating that the deployments are to be shared. Shared J2EE libraries must be targeted to the same WebLogic Server instances you want to deploy applications that reference the libraries. If you try to deploy a referencing application to a server instance that has not registered a required library, deployment of the referencing application fails. See Registering Libraries with WebLogic Server in *Deploying Applications to WebLogic Server* for more information.

See Install a Shared J2EE Library for detailed instructions on installing (deploying) a shared J2EE library using the Administration Console. See Target a Shared J2EE Library to a Server or Cluster for instructions on using the Administration Console to target the library to the server or cluster to which the application that is referencing the library is also targeted.

If you use the `wldeploy` Ant task as part of your iterative development process, use the `library`, `libImplVer`, and `libSpecVer` attributes to deploy a shared J2EE library. See Appendix B, "wldeploy Ant Task Reference," for details and examples.

After registering a shared J2EE library, you can deploy applications and archives that depend on the library. Dependent applications can be deployed only if the target servers have registered all required libraries, and the registered deployments meet the version requirements of the application or archive. See Deploying Applications that Reference Libraries in *Deploying Applications to WebLogic Server* for more information.

Web Application Shared J2EE Library Information

In general, this topic discusses shared J2EE libraries that can be referenced only by Enterprise Applications. You can also create libraries that can be referenced only by another Web application. The functionality is very similar to application libraries, although the method of referencing them is slightly different.

Note: For simplicity, this section uses the term *Web application library* when referring to a shared J2EE library that is referenced only by another Web application.

In particular:

- Web application libraries can only be referenced by other Web applications.
- Rather than update the `weblogic-application.xml` file, Web applications reference Web application libraries by updating the `weblogic.xml` deployment descriptor file. The elements are almost same as those described in "Referencing Shared J2EE Libraries in an Enterprise Application" on page 8-11; the only difference is that the `<context-root>` child element of `<library-ref>` is ignored in this case.
- You cannot reference any other type of shared J2EE library (EJB, Enterprise application, or plain JAR file) from the `weblogic.xml` deployment descriptor file of a Web Application.

Other than these differences in how they are referenced, the way to create, package, and deploy a Web application library is the same as that of a standard shared J2EE library.

Accessing Registered Shared J2EE Library Information with LibraryRuntimeMBean

Each deployed shared J2EE library is represented by a `LibraryRuntimeMBean`. You can use this MBean to obtain information about the library itself, such as its name or version. You can also obtain the `ApplicationRuntimeMBean`s associated with deployed applications. `ApplicationRuntimeMBean` provides two methods to access the libraries that the application is using:

- `getLibraryRuntimes()` returns the shared J2EE libraries referenced in the `weblogic-application.xml` file.
- `getOptionalPackageRuntimes()` returns the optional packages referenced in the manifest file.

For more information, see the WebLogic Server 9.0 API Reference.

Order of Precedence of Modules When Referencing Shared J2EE Libraries

When an Enterprise Application references one or more shared J2EE libraries, and the application is deployed to WebLogic Server, the server internally merges the information in the `weblogic-application.xml` file of the referencing Enterprise Application with the information in the deployment descriptors of the referenced libraries. The order in which this happens is as follows:

1. When the Enterprise Application is deployed, WebLogic Server reads its `weblogic-application.xml` deployment descriptor.
2. WebLogic Server reads the deployment descriptors of any referenced shared J2EE libraries. Depending on the type of library (Enterprise Application, EJB, or Web application), the read file might be `weblogic-application.xml`, `weblogic.xml`, `weblogic-ejb-jar.xml`, and so on.
3. WebLogic Server first merges the referenced shared J2EE library deployment descriptors (in the order in which they are referenced, one at a time) and then merges the `weblogic-application.xml` file of the referencing Enterprise application on top of the library descriptor files.

As a result of the way the descriptor files are merged, the elements in the descriptors of the shared J2EE libraries referenced first in the `weblogic-application.xml` file have precedence over the ones listed last. The elements of the Enterprise application's descriptor itself have precedence over all elements in the library descriptors.

For example, assume that an Enterprise application called `myApp` references two shared J2EE libraries (themselves packaged as Enterprise applications): `myLibA` and `myLibB`, in that order. Both the `myApp` and `myLibA` applications include an EJB module called `myEJB`, and both the `myLibA` and `myLibB` applications include an EJB module called `myOtherEJB`.

Further assume that once the `myApp` application is deployed, a client invokes, via the `myApp` application, the `myEJB` module. In this case, WebLogic Server actually invokes the EJB in the `myApp` application (rather than the one in `myLibA`) because modules in the *referencing* application have higher precedence over modules in the *referenced* applications. If a client invokes the `myOtherEJB` EJB, then WebLogic Server invokes the one in `myLibA`, because the library is referenced first in the `weblogic-application.xml` file of `myApp`, and thus has precedence over the EJB with the same name in the `myLibB` application.

Best Practices for Using Shared J2EE Libraries

Keep in mind these best practices when developing shared J2EE libraries and optional packages:

- Use shared J2EE Libraries when you want to share one or more J2EE modules (EJBs, Web Applications, Enterprise Applications, or plain Java classes) with multiple Enterprise Applications.
- If you need to deploy a standalone J2EE module, such as an EJB JAR file, as a shared J2EE library, package the module within an Enterprise Application. Doing so avoids potential URI conflicts, because the library URI of a standalone module is derived from the deployment name.
- If you choose to deploy a shared J2EE library as a standalone J2EE module, always specify a known deployment name during deployment and use that name as the URI in referencing applications.
- Use optional packages when multiple J2EE archive files need to share a set of Java classes.
- If you have a set of classes that must be available to applications in an entire domain, and you do not frequently update those classes (for example, if you need to share 3rd party classes in a domain), use the domain `/lib` subdirectory rather than using shared J2EE libraries or optional packages. Classes in the `/lib` subdirectory are added to the system classpath at server start-up time.
- Always specify a specification version and implementation version, even if you do not intend to enforce version requirements with dependent applications. Specifying versions for shared J2EE libraries enables you to deploy multiple versions of the shared files for testing.
- Always specify an `Extension-Name` value for each shared J2EE library. If you do not specify an extension name, one is derived from the deployment name of the library. Default deployment names are different for archive and exploded archive deployments, and they can be set to arbitrary values in the deployment command
- When developing a Web Application for deployment as a shared J2EE library, use a unique context root. If the context root conflicts with the context root in a dependent J2EE application, use the `context-root` element in the EAR's `weblogic-application.xml` deployment descriptor to override the library's context root.
- Package shared J2EE libraries as archive files for delivery to Administrators or deployers in your organization. Deploy libraries from exploded archive directories during development to allow for easy updates and repeated redeployments.

- Deploy shared J2EE libraries to all WebLogic Server instances on which you want to deploy dependent applications and archives. If a library is not registered with a server instance on which you want to deploy a referencing application, deployment of the referencing application fails.

CHAPTER 9

Programming Application Lifecycle Events

The following sections describe how to create applications that respond to WebLogic Server application lifecycle events:

Warning: Application-scoped startup and shutdown classes have been deprecated in this release of WebLogic Server. The information in this chapter about startup and shutdown classes is provided only for backwards compatibility. Instead, you should use lifecycle listener events in your applications.

Understanding Application Lifecycle Events

Application lifecycle listener events provide handles on which developers can control behavior during deployment, undeployment, and redeployment. This section discusses how you can use the application lifecycle listener events.

Four application lifecycle events are provided with WebLogic Server, which can be used to extend listener, shutdown, and startup classes. These include:

- Listeners—attachable to any event. Possible methods for Listeners are:

  ```
  public void preStart(ApplicationLifecycleEvent evt) {}
  ```

  – The preStart event is the beginning of the prepare phase, or the start of the application deployment process.)

  ```
  public void postStart(ApplicationLifecycleEvent evt) {}
  ```

  – The postStart event is the end of the activate phase, or the end of the application deployment process. The application is deployed.

  ```
  public void preStop(ApplicationLifecycleEvent evt) {}
  ```

  – The preStop event is the beginning of the deactivate phase, or the start of the application removal or undeployment process.

  ```
  public void postStop(ApplicationLifecycleEvent evt) {}
  ```

  – The postStop event is the end of the remove phase, or the end of the application removal or undeployment process.

- Shutdown classes only get postStop events.

  Warning: Application-scoped shutdown classes have been deprecated in this release of WebLogic Server. Use lifecycle listeners instead.

- Startup classes only get preStart events.

  Warning: Application-scoped shutdown classes have been deprecated in this release of WebLogic Server. Use lifecycle listeners instead.

Note: For Startup and Shutdown classes, you only implement a `main{}` method. If you implement any of the methods provided for Listeners, they are ignored.

Note: No `remove{}` method is provided in the ApplicationLifecycleListener, because the events are only fired at startup time during deployment (prestart and poststart) and shutdown during undeployment (prestop and poststop).

Registering Events in weblogic-application.xml

In order to use these events, you must register them in the `weblogic-application.xml` deployment descriptor. See "Application Deployment Descriptor Elements." Define the following elements:

- `listener`—Used to register user defined application lifecycle listeners. These are classes that extend the abstract base class `weblogic.application.ApplicationLifecycleListener`.
- `shutdown`—Used to register user-defined shutdown classes.
- `startup`—Used to register user-defined startup classes.

Programming Basic Lifecycle Listener Functionality

You create a listener by extending the abstract class (provided with WebLogic Server) `weblogic.application.ApplicationLifecycleListener`. The container then searches for your listener.

You override the following methods provided in the WebLogic Server `ApplicationLifecycleListener` abstract class to extend your application and add any required functionality:

- preStart{}
- postStart{}
- preStop{}
- postStop{}

Listing 9-1 illustrates how you override the `ApplicationLifecycleListener`. In this example, the public class `MyListener` extends `ApplicationLifecycleListener`.

Listing 9-1 MyListener

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyListener extends ApplicationLifecycleListener {
  public void preStart(ApplicationLifecycleEvent evt) {
```

```
      System.out.println
      ("MyListener(preStart) -- we should always see you..");
    } // preStart
  public void postStart(ApplicationLifecycleEvent evt) {
      System.out.println
      ("MyListener(postStart) -- we should always see you..");
    } // postStart
  public void preStop(ApplicationLifecycleEvent evt) {
      System.out.println
      ("MyListener(preStop) -- we should always see you..");
    } // preStop
  public void postStop(ApplicationLifecycleEvent evt) {
      System.out.println
      ("MyListener(postStop) -- we should always see you..");
    } // postStop
   public static void main(String[] args) {
      System.out.println
      ("MyListener(main): in main .. we should never see you..");
    } // main
}
```

Listing 9-2 illustrates how you implement the shutdown class. The shutdown class is attachable to preStop and postStop events. In this example, the public class `MyShutdown` extends `ApplicationLifecycleListener`.

Listing 9-2 MyShutdown

```
import weblogic.application.ApplicationLifecycleListener;
```

```
import weblogic.application.ApplicationLifecycleEvent;
public class MyShutdown extends ApplicationLifecycleListener {
   public static void main(String[] args) {
     System.out.println
     ("MyShutdown(main): in main .. should be for post-stop");
   } // main
}
```

Listing 9-3 illustrates how you implement the startup class. The startup class is attachable to preStart and postStart events. In this example, the public class `MyStartup` extends `ApplicationLifecycleListener`.

Listing 9-3 MyStartup

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyStartup extends ApplicationLifecycleListener {
   public static void main(String[] args) {
     System.out.println
     ("MyStartup(main): in main .. should be for pre-start");
   } // main
}
```

Examples of Configuring Lifecycle Events with and without the URI Parameter

The following examples illustrate how you configure application lifecycle events in the `weblogic-application.xml` deployment descriptor file. The URI parameter is not required. You can place classes anywhere in the application `$CLASSPATH`. However, you must ensure that

the class locations are defined in the `$CLASSPATH`. You can place listeners in `APP-INF/classes` or `APP-INF/lib`, if these directories are present in the EAR. In this case, they are automatically included in the `$CLASSPATH`.

The following example illustrates how you configure application lifecycle events using the URI parameter. In this case, the archive `foo.jar` contains the classes and exists at the top level of the EAR file. For example: `myEar/foo.jar`

Listing 9-4 Configuring Application Lifecycle Events Using the URI Parameter

```
<listener>
     <listener-class>MyListener</listener-class>
     <listener-uri>foo.jar</listener-uri>
</listener>
<startup>
     <startup-class>MyStartup</startup-class>
     <startup-uri>foo.jar</startup-uri>
</startup>
<shutdown>
     <shutdown-class>MyShutdown</shutdown-class>
     <shutdown-uri>foo.jar</shutdown-uri>
</shutdown>
```

The following example illustrates how you configure application lifecycle events without using the URI parameter.

Listing 9-5 Configuring Application Lifecycle Events without Using the URI Parameter

```
<listener>
     <listener-class>MyListener</listener-class>
 </listener>
```

```
<startup>
     <startup-class>MyStartup</startup-class>
</startup>
<shutdown>
     <shutdown-class>MyShutdown</shutdown-class>
</shutdown>
```

Understanding Application Lifecycle Event Behavior During Re-deployment

Application lifecycle events are only triggered if a full re-deployment of the application occurs. During a full re-deployment of the application—provided the application lifecycle events have been registered—the application lifecycle first commences the shutdown sequence, next re-initializes its classes, and then performs the startup sequence.

For example, if your listener is registered for the full application lifecycle set of events (preStart, postStart, preStop, postStop), during a full re-deployment, you see the following sequence of events:

1. `preStop{}`
2. `postStop{}`
3. Initialization takes place. (Unless you have set debug flags, you do not see the initialization.)
4. `preStart{}`
5. `postStart{}`

CHAPTER 10

Programming Context Propagation

The following sections describe how to use the context propagation APIs in your applications:

Understanding Context Propagation

Context propagation allows programmers to associate information with an application which is then carried along with every request. Furthermore, downstream components can add or modify this information so that it can be carried back to the originator. Context propagation is also known as *work areas*, *work contexts*, or *application transactions*.

Common use-cases for context propagation are any type of application in which information needs to be carried outside the application, rather than the information being an integral part of the application. Examples of these use cases include diagnostics monitoring, application transactions, and application load-balancing. Keeping this sort of information outside of the application keeps the application itself clean with no extraneous API usage and also allows the addition of information to read-only components, such as 3rd party components.

Programming context propagation has two parts: first you code the client application to create a `WorkContextMap` and `WorkContext`, and then add user data to the context, and then you code the invoked application itself to get and possibly use this data. The invoked application can be of

any type: EJB, Web Service, servlet, JMS topic or queue, and so on. See "Programming Context Propagation: Main Steps" on page 10-3 for details.

The WebLogic context propagation APIs are in the `weblogic.workarea` package. The following table describes the main interfaces and classes.

Table 10-1 Interfaces and classes of the WebLogic Context Propagation API

Interface or Class	Description
`WorkContextMap` Interface	Main context propagation interface used to tag applications with data and propagate that information via application requests. `WorkContextMaps` is part of the client or application's JNDI environment and can be accessed through JNDI by looking up the name `java:comp/WorkContextMap`.
`WorkContext` Interface	Interface used for marshaling and unmarshaling the user data that is passed along with an application. This interface has four implementing classes for marshaling and unmarshaling the following types of data: simple 8-bit ASCII contexts (`AsciiWorkContext`), long contexts (`LongWorkContext`), Serializable context (`SerializableWorkContext`), and String contexts (`StringWorkContext`). `WorkContext` has one subinterface, `PrimitiveWorkContext`, used to specifically marshal and unmarshal a single primitive data item.
`WorkContextOutput/Input` Interfaces	Interfaces representing primitive streams used for marshaling and unmarshaling, respectively, `WorkContext` implementations.
`PropagationMode` Interface	Defines the propagation properties of `WorkContexts`. Specifies whether the WorkContext is propagated locally, across threads, across RMI invocations, across JMS queues and topics, or across SOAP messages. If not specified, default is to propagate data across remote and local calls in the same thread.
`PrimitiveContextFactory` Class	Convenience class for creating `WorkContexts` that contain only primitive data.

For the complete API documentation about context propagation, see the `weblogic.workarea` Javadocs.

Programming Context Propagation: Main Steps

The following procedure describes the high-level steps to use context propagation in your application. It is assumed in the procedure that you have already set up your iterative development environment and have an existing client and application that you want to update to use context propagation by using the `weblogic.workarea` API.

1. Update your client application to create the `WorkContextMap` and `WorkContext` objects and then add user data to the context.

   See "Programming Context Propagation in a Client" on page 10-3.

2. If your client application is standalone (rather than running in a J2EE component deployed to WebLogic Server), ensure that its CLASSPATH includes the J2EE application client, also called the *thin client*.

   See Programming Stand-Alone Clients.

3. Update your application (EJB, Web Service, servlet, and so on) to also create a `WorkContextMap` and then get the context and user data that you added from the client application.

   See "Programming Context Propagation in an Application" on page 10-5.

Programming Context Propagation in a Client

The following sample Java code shows a standalone Java client that invokes a Web Service; the example also shows how to use the `weblogic.workarea.*` context propagation APIs to associate user information with the invoke. The code relevant to context propagation is shown in bold and explained after the example.

For the complete API documentation about context propagation, see the `weblogic.workarea` Javadocs.

Note: See Programming Web Services for WebLogic Server for information on creating Web Services and client applications that invoke them.

```
package examples.workarea.client;

import java.rmi.RemoteException;
```

```
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import weblogic.workarea.WorkContextMap;
import weblogic.workarea.WorkContext;
import weblogic.workarea.PrimitiveContextFactory;
import weblogic.workarea.PropagationMode;
import weblogic.workarea.PropertyReadOnlyException;

/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of the WorkArea Web service.
 *
 * @author Copyright (c) 2004 by BEA Systems. All Rights Reserved.
 */

public class Main {

  public final static String SESSION_ID= "session_id_key";

  public static void main(String[] args)
      throws ServiceException, RemoteException, NamingException,
PropertyReadOnlyException{

    WorkAreaService service = new WorkAreaService_Impl(args[0] + "?WSDL");
    WorkAreaPortType port = service.getWorkAreaPort();

    WorkContextMap map = (WorkContextMap)new
InitialContext().lookup("java:comp/WorkContextMap");

    WorkContext stringContext = PrimitiveContextFactory.create("A String
Context");

    // Put a string context
    map.put(SESSION_ID, stringContext, PropagationMode.SOAP);

    try {
      String result = null;
      result = port.sayHello("Hi there!");
      System.out.println( "Got result: " + result );
    } catch (RemoteException e) {
      throw e;
    }
  }
}
```

In the preceding example:

- The following code shows how to import the needed `weblogic.workarea.*` classes, interfaces, and exceptions:

```
import weblogic.workarea.WorkContextMap;
import weblogic.workarea.WorkContext;
import weblogic.workarea.PrimitiveContextFactory;
import weblogic.workarea.PropagationMode;
import weblogic.workarea.PropertyReadOnlyException;
```

- The following code shows how to create a `WorkContextMap` by doing a JNDI lookup of the context propagation-specific JNDI name `java:comp/WorkContextMap`:

```
WorkContextMap map = (WorkContextMap)
   new InitialContext().lookup("java:comp/WorkContextMap");
```

- The following code shows how to create a `WorkContext` by using the `PrimitiveContextFactory`. In this example, the `WorkContext` consists of the simple String value `A String Context`. This String value is the user data that is passed to the invoked Web Service.

```
WorkContext stringContext =
    PrimitiveContextFactory.create("A String Context");
```

- Finally, the following code shows how to add the context data, along with the key `SESSION_ID`, to the `WorkContextMap` and associate it with the current thread. The `PropagationMode.SOAP` constant specifies that the propagation happens over SOAP messages; this is because the client is invoking a Web Service.

```
map.put(SESSION_ID, stringContext, PropagationMode.SOAP);
```

Programming Context Propagation in an Application

The following sample Java code shows a simple Java Web Service (JWS) file that implements a Web Service. The JWS file also includes context propagation code to get the user data that is associated with the invoke of the Web Service. The code relevant to context propagation is shown in bold and explained after the example.

For the complete API documentation about context propagation, see the `weblogic.workarea` Javadocs.

Note: See Programming Web Services for WebLogic Server for information on creating Web Services and client applications that invoke them.

```
package examples.workarea;

import javax.naming.InitialContext;
```

```
// Import the Context Propagation classes

import weblogic.workarea.WorkContextMap;
import weblogic.workarea.WorkContext;

import javax.jws.WebMethod;
import javax.jws.WebService;

import weblogic.jws.WLHttpTransport;

@WebService(name="WorkAreaPortType",
            serviceName="WorkAreaService",
             targetNamespace="http://example.org")

@WLHttpTransport(contextPath="workarea",
                 serviceUri="WorkAreaService",
                  portName="WorkAreaPort")

/**
 * This JWS file forms the basis of simple WebLogic
 * Web Service with a single operation: sayHello
 *
 */

public class WorkAreaImpl {

  public final static String SESSION_ID = "session_id_key";

  @WebMethod()
  public String sayHello(String message) {

   try {

    WorkContextMap map = (WorkContextMap) new
InitialContext().lookup("java:comp/WorkContextMap");

    WorkContext localwc = map.get(SESSION_ID);

    System.out.println("local context: " + localwc);
    System.out.println("sayHello: " + message);

    return "Here is the message: '" + message + "'";

   } catch (Throwable t) {

    return "error";

   }

  }

}
```

In the preceding example:

- The following code shows how to import the needed context propagation APIs; in this case, only the WorkContextMap and WorkContext interfaces are needed:

```
import weblogic.workarea.WorkContextMap;
import weblogic.workarea.WorkContext;
```

- The following code shows how to create a `WorkContextMap` by doing a JNDI lookup of the context propagation-specific JNDI name `java:comp/WorkContextMap`:

```
WorkContextMap map = (WorkContextMap)
   new InitialContext().lookup("java:comp/WorkContextMap");
```

- The following code shows how to get context's user data from the current `WorkContextMap` using a key; in this case, the key is the same one that the client application set when it invoked the Web Service: `SESSION_ID`:

```
WorkContext localwc = map.get(SESSION_ID);
```

CHAPTER 11

Programming JavaMail with WebLogic Server

The following sections contains information on additional WebLogic Server programming topics:

Overview of Using JavaMail with WebLogic Server Applications

WebLogic Server includes the JavaMail API version 1.3 reference implementation from Sun Microsystems. Using the JavaMail API, you can add email capabilities to your WebLogic Server applications. JavaMail provides access from Java applications to Internet Message Access Protocol (IMAP)- and Simple Mail Transfer Protocol (SMTP)-capable mail servers on your network or the Internet. It does not provide mail server functionality; you must have access to a mail server to use JavaMail.

Complete documentation for using the JavaMail API is available on the JavaMail page on the Sun Web site at http://java.sun.com/products/javamail/index.html. This section describes how you can use JavaMail in the WebLogic Server environment.

The `weblogic.jar` file contains the `javax.mail` and `javax.mail.internet` packages from Sun. `weblogic.jar` also contains the Java Activation Framework (JAF) package, which JavaMail requires.

The `javax.mail` package includes providers for Internet Message Access protocol (IMAP) and Simple Mail Transfer Protocol (SMTP) mail servers. Sun has a separate POP3 provider for JavaMail, which is not included in `weblogic.jar`. You can download the POP3 provider from Sun and add it to the WebLogic Server classpath if you want to use it.

Understanding JavaMail Configuration Files

JavaMail depends on configuration files that define the mail transport capabilities of the system. The `weblogic.jar` file contains the standard configuration files from Sun, which enable IMAP and SMTP mail servers for JavaMail and define the default message types JavaMail can process.

Unless you want to extend JavaMail to support additional transports, protocols, and message types, you do not have to modify any JavaMail configuration files. If you do want to extend JavaMail, download JavaMail from Sun and follow Sun's instructions for adding your extensions. Then add your extended JavaMail package in the WebLogic Server classpath *in front of* `weblogic.jar`.

Configuring JavaMail for WebLogic Server

To configure JavaMail for use in WebLogic Server, you create a mail session in the WebLogic Server Administration Console. This allows server-side modules and applications to access JavaMail services with JNDI, using Session properties you preconfigure for them. For example, by creating a mail session, you can designate the mail hosts, transport and store protocols, and the default mail user in the Administration Console so that modules that use JavaMail do not have to

set these properties. Applications that are heavy email users benefit because the mail session creates a single `javax.mail.Session` object and makes it available via JNDI to any module that needs it.

For information on using the Administration Console to create a mail session, see Configure access to JavaMail in the *Administration Console Online Help*.

You can override any properties set in the mail session in your code by creating a `java.util.Properties` object containing the properties you want to override. See "Sending Messages with JavaMail" on page 11-3. Then, after you look up the mail session object in JNDI, call the `Session.getInstance()` method with your `Properties` object to get a customized Session.

Sending Messages with JavaMail

Here are the steps to send a message with JavaMail from within a WebLogic Server module:

1. Import the JNDI (naming), JavaBean Activation, and JavaMail packages. You will also need to import `java.util.Properties`:

   ```
   import java.util.*;
   import javax.activation.*;
   import javax.mail.*;
   import javax.mail.internet.*;
   import javax.naming.*;
   ```

2. Look up the Mail Session in JNDI:

   ```
   InitialContext ic = new InitialContext();
   Session session = (Session) ic.lookup("myMailSession");
   ```

3. If you need to override the properties you set for the Session in the Administration Console, create a `java.util.Properties` object and add the properties you want to override. Then call `getInstance()` to get a new Session object with the new properties.

   ```
   Properties props = new Properties();
   props.put("mail.transport.protocol", "smtp");
   props.put("mail.smtp.host", "mailhost");
   // use mail address from HTML form for from address
   props.put("mail.from", emailAddress);
   Session session2 = session.getInstance(props);
   ```

4. Construct a `MimeMessage`. In the following example, *`to`*, *`subject`*, and *`messageTxt`* are String variables containing input from the user.

   ```
   Message msg = new MimeMessage(session2);
   msg.setFrom();
   ```

```
msg.setRecipients(Message.RecipientType.TO,
                  InternetAddress.parse(to, false));
msg.setSubject(subject);
msg.setSentDate(new Date());
// Content is stored in a MIME multi-part message
// with one body part
MimeBodyPart mbp = new MimeBodyPart();
mbp.setText(messageTxt);

Multipart mp = new MimeMultipart();
mp.addBodyPart(mbp);
msg.setContent(mp);
```

5. Send the message.

```
Transport.send(msg);
```

The JNDI lookup can throw a `NamingException` on failure. JavaMail can throw a `MessagingException` if there are problems locating transport classes or if communications with the mail host fails. Be sure to put your code in a try block and catch these exceptions.

Reading Messages with JavaMail

The JavaMail API allows you to connect to a message store, which could be an IMAP server or POP3 server. Messages are stored in folders. With IMAP, message folders are stored on the mail server, including folders that contain incoming messages and folders that contain archived messages. With POP3, the server provides a folder that stores messages as they arrive. When a client connects to a POP3 server, it retrieves the messages and transfers them to a message store on the client.

Folders are hierarchical structures, similar to disk directories. A folder can contain messages or other folders. The default folder is at the top of the structure. The special folder name INBOX refers to the primary folder for the user, and is within the default folder. To read incoming mail, you get the default folder from the store, and then get the INBOX folder from the default folder.

The API provides several options for reading messages, such as reading a specified message number or range of message numbers, or pre-fetching specific parts of messages into the folder's cache. See the JavaMail API for more information.

Here are steps to read incoming messages on a POP3 server from within a WebLogic Server module:

1. Import the JNDI (naming), JavaBean Activation, and JavaMail packages. You will also need to import `java.util.Properties`:

```
import java.util.*;
import javax.activation.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.naming.*;
```

2. Look up the Mail Session in JNDI:

```
InitialContext ic = new InitialContext();
Session session = (Session) ic.lookup("myMailSession");
```

3. If you need to override the properties you set for the Session in the Administration Console, create a `Properties` object and add the properties you want to override. Then call `getInstance()` to get a new Session object with the new properties:

```
Properties props = new Properties();
props.put("mail.store.protocol", "pop3");
props.put("mail.pop3.host", "mailhost");
Session session2 = session.getInstance(props);
```

4. Get a `Store` object from the Session and call its `connect()` method to connect to the mail server. To authenticate the connection, you need to supply the mailhost, username, and password in the connect method:

```
Store store = session.getStore();
store.connect(mailhost, username, password);
```

5. Get the default folder, then use it to get the INBOX folder:

```
Folder folder = store.getDefaultFolder();
folder = folder.getFolder("INBOX");
```

6. Read the messages in the folder into an array of Messages:

```
Message[] messages = folder.getMessages();
```

7. Operate on messages in the Message array. The Message class has methods that allow you to access the different parts of a message, including headers, flags, and message contents.

Reading messages from an IMAP server is similar to reading messages from a POP3 server. With IMAP, however, the JavaMail API provides methods to create and manipulate folders and transfer messages between them. If you use an IMAP server, you can implement a full-featured, Web-based mail client with much less code than if you use a POP3 server. With POP3, you must provide code to manage a message store via WebLogic Server, possibly using a database or file system to represent folders.

CHAPTER 12

Threading and Clustering Topics

The following sections contain information on additional WebLogic Server programming topics:

Using Threads in WebLogic Server

WebLogic Server is a sophisticated, multi-threaded application server and it carefully manages resource allocation, concurrency, and thread synchronization for the modules it hosts. To obtain the greatest advantage from WebLogic Server's architecture, construct your application modules created according to the standard J2EE APIs.

In most cases, avoid application designs that require creating new threads in server-side modules:

- Applications that create their own threads do not scale well. Threads in the JVM are a limited resource that must be allocated thoughtfully. Your applications may break or cause WebLogic Server to thrash when the server load increases. Problems such as deadlocks and thread starvation may not appear until the application is under a heavy load.
- Multithreaded modules are complex and difficult to debug. Interactions between application-generated threads and WebLogic Server threads are especially difficult to anticipate and analyze.

In some situations, creating threads may be appropriate, in spite of these warnings. For example, an application that searches several repositories and returns a combined result set can return results sooner if the searches are done asynchronously using a new thread for each repository instead of synchronously using the main client thread.

If you must use threads in your application code, create a pool of threads so that you can control the number of threads your application creates. Like a JDBC connection pool, you allocate a given number of threads to a pool, and then obtain an available thread from the pool for your runnable class. If all threads in the pool are in use, wait until one is returned. A thread pool helps avoid performance issues and allows you to optimize the allocation of threads between WebLogic Server execution threads and your application.

Be sure you understand where your threads can deadlock and handle the deadlocks when they occur. Review your design carefully to ensure that your threads do not compromise the security system.

To avoid undesirable interactions with WebLogic Server threads, do not let your threads call into WebLogic Server modules. For example, do not use enterprise beans or servlets from threads that you create. Application threads are best used for independent, isolated tasks, such as conversing with an external service with a TCP/IP connection or, with proper locking, reading or writing to files. A short-lived thread that accomplishes a single purpose and ends (or returns to the thread pool) is less likely to interfere with other threads.

Avoid creating daemon threads in modules that are packaged in applications deployed on WebLogic Server. When you create a daemon thread in an application module such as a Servlet,

you will not be able to redeploy the application because the daemon thread created in the original deployment will remain running.

Be sure to test multithreaded code under increasingly heavy loads, adding clients even to the point of failure. Observe the application performance and WebLogic Server behavior and then add checks to prevent failures from occurring in production.

Programming Applications for WebLogic Server Clusters

JSPs and Servlets that will be deployed to a WebLogic Server cluster must observe certain requirements for preserving session data. See "Requirements for HTTP Session State Replication" in *Using WebLogic Server Clusters* for more information.

EJBs deployed in a WebLogic Server cluster have certain restrictions based on EJB type. See "Understanding WebLogic Enterprise JavaBeans" in *Programming WebLogic Enterprise JavaBeans* for information about the capabilities of different EJB types in a cluster. EJBs can be deployed to a cluster by setting clustering properties in the EJB deployment descriptor.

If you are developing either EJBs or custom RMI objects for deployment in a cluster, also refer to "Using WebLogic JNDI in a Clustered Environment" in *Programming WebLogic JNDI* to understand the implications of binding clustered objects in the JNDI tree.

APPENDIX A

Enterprise Application Deployment Descriptor Elements

The following sections describe Enterprise application deployment descriptors: `application.xml` (a J2EE standard deployment descriptor) and `weblogic-application.xml` (a WebLogic-specific application deployment descriptor).

The `weblogic-application.xml` file is optional if you are not using any WebLogic Server extensions.

weblogic-application.xml Deployment Descriptor Elements

The following sections describe the many of the individual elements that are defined in the weblogic-application.xml Schema. The `weblogic-application.xml` file is the BEA WebLogic Server-specific deployment descriptor extension for the `application.xml` deployment descriptor from Sun Microsystems. This is where you configure features such as shared J2EE libraries referenced in the application and EJB caching.

The file is located in the `META-INF` subdirectory of the application archive. The following sections describe elements that can appear in the file.

weblogic-application

The `weblogic-application` element is the root element of the application deployment descriptor.

The following table describes the elements you can define within a `weblogic-application` element.

Element	Required?	Maximum Number In File	Description
`<ejb>`	Optional	1	Contains information that is specific to the EJB modules that are part of a WebLogic application. Currently, one can use the `ejb` element to specify one or more application level caches that can be used by the application's entity beans. For more information on the elements you can define within the `ejb` element, refer to "ejb" on page A-10.
`<xml>`	Optional	1	Contains information about parsers and entity mappings for XML processing that is specific to this application. For more information on the elements you can define within the `xml` element, refer to "xml" on page A-15.
`<jdbc-connection-pool>`	Optional	Unbounded	Zero or more. Specifies an application-scoped JDBC connection pool. For more information on the elements you can define within the `jdbc-connection-pool` element, refer to "jdbc-connection-pool" on page A-17.
`<security>`	Optional	1	Specifies security information for the application. For more information on the elements you can define within the `security` element, refer to "security" on page A-32.

Element	Required?	Maximum Number In File	Description

`<application-param>`	Optional	Unbounded	Zero or more. Used to specify un-typed parameters that affect the behavior of container instances related to the application. The parameters listed here are currently supported. Also, these parameters in `weblogic-application.xml` can determine the default encoding to be used for requests and for responses.
			• `webapp.encoding.default`—Can be set to a string representing an encoding supported by the JDK. If set, this defines the default encoding used to process servlet requests and servlet responses. This setting is ignored if `webapp.encoding.usevmdefault` is set to `true`. This value is also overridden for request streams by the `input-charset` element of `weblogic.xml`.
			• `webapp.encoding.usevmdefault`—Can be set to `true` or `false`. If `true`, the system property `file.encoding` is used to define the default encoding.
			The following parameter is used to affect the behavior of Web applications that are contained in this application.
			• `webapp.getrealpath.accept_context_path`—This is a compatibility switch that may be set to `true` or `false`. If set to `true`, the context path of Web applications is allowed in calls to the servlet API `getRealPath`.
			Example:
			`<application-param>`
			`<param-name>webapp.encoding.default </param-name>`
			`<param-value>UTF8</param-value>`
			`</application-param>`
			For more information on the elements you can define within the `application-param` element, refer to "application-param" on page A-32.

Element	Required?	Maximum Number In File	Description

`<classloader-structure>`	Optional	Unbounded	A `classloader-structure` element allows you to define the organization of classloaders for this application. The declaration represents a tree structure that represents the classloader hierarchy and associates specific modules with particular nodes. A module's classes are loaded by the classloader that its associated with this element.
			Example:
			`<classloader-structure>`
			<module-ref>
			<module-uri>ejb1.jar</module-uri>
			</module-ref>
			`</classloader-structure>`
			`<classloader-structure>`
			<module-ref>
			<module-uri>ejb2.jar</module-uri>
			</module-ref>
			`</classloader-structure>`
			For more information on the elements you can define within the `classloader-structure` element, refer to "classloader-structure" on page A-33.

`<listener>`	Optional	Unbounded	Zero or more. Used to register user defined application lifecycle listeners. These are classes that extend the abstract base class `weblogic.application.ApplicationLifecycleListener`. For more information on the elements you can define within the `listener` element, refer to "listener" on page A-33.

Element	Required?	Maximum Number In File	Description
<startup>	Optional	Unbounded	Zero or more. Used to register user-defined startup classes. For more information on the elements you can define within the startup element, refer to "startup" on page A-34. **Note:** Application-scoped startup and shutdown classes have been deprecated in this release of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Chapter 9, "Programming Application Lifecycle Events."
<shutdown>	Optional	Unbounded	Zero or more. Used to register user defined shutdown classes. For more information on the elements you can define within the shutdown element, refer to "shutdown" on page A-34. **Note:** Application-scoped startup and shutdown classes have been deprecated in this release of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Chapter 9, "Programming Application Lifecycle Events."

Element	Required?	Maximum Number In File	Description

`<module>`	Optional	Unbounded	Represents a single WebLogic application module, such as a JMS or JDBC module.
			This element has the following child elements:
			• `name`—The name of the module.
			• `type`—The type of module. Valid values are JMS, JDBC, or Interception.
			• `path`—The path of the XML file that fully describes the module, relative to the root of the Enterprise application.
			The following example shows how to specify a JMS module called `Workflows`, fully described by the XML file `jms/Workflows-jms.xml`:
			`<module>`
			`<name>Workflows</name>`
			`<type>JMS</type>`
			`<path>jms/Workflows-jms.xml</path>`
			`</module>`

`<library-ref>`	Optional	Unbounded	A reference to a shared J2EE library. For more information on the elements you can define within the `library` element, refer to "library" on page A-40.
`<fair-share-request>`	Optional	Unbounded	Specifies a fair share request class, which is a type of Work Manager request class. In particular, a fair share request class specifies the average percentage of thread-use time required to process requests. The `<fair-share-request>` element can take the following child elements: • `name`—The name of the fair share request class. • `fair-share`—An integer representing the average percentage of thread-use time. See Using Work Managers to Optimize Schedule Work.

Element	Required?	Maximum Number In File	Description
`<response-time-request>`	Optional	Unbounded	Specifies a response time request class, which is a a type of Work manager class. In particular, a response time request class specifies a response time goal in milliseconds. The `<response-time-request>` element can take the following child elements: • `name`—The name of the response time request class. • `goal-ms`—The integer response time goal. See Using Work Managers to Optimize Schedule Work.
`<context-request>`	Optional	Unbounded	Specifies a context request class, which is a a type of Work manager class. In particular, a context request class assigns request classes to requests based on context information, such as the current user or the current user's group. The `<context-request>` element can take the following child elements: • `name`—The name of the context request class. • `context-case`—An element that describes the context. The `<context-case>` element can itself take the following child elements: • `user-name` or `group-name`—The user or group to which the context applies. • `request-class-name`—The name of the request class. See Using Work Managers to Optimize Schedule Work.

Element	Required?	Maximum Number In File	Description
<max-threads-constraint>	Optional	Unbounded	Specifies a max-threads-constraint Work Manager constraint. A Work Manager constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests. The max-threads constraint limits the number of concurrent threads executing requests from the constrained work set. The <max-threads-constraint> element can take the following child elements: • name—The name of the max-thread-constaint constraint. • Either count or pool-name—The integer maximum number of concurrent threads, or the name of a connection pool which determines the maximum. See Using Work Managers to Optimize Schedule Work.
<min-threads-constraint>	Optional	Unbounded	Specifies a min-threads-constraint Work Manager constraint. A Work Manager constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests. The min-threads constraint guarantees a number of threads the server will allocate to affected requests to avoid deadlocks. The <min-threads-constraint> element can take the following child elements: • name—The name of the min-thread-constaint constraint. • count—The integer minimum number of threads. See Using Work Managers to Optimize Schedule Work.

Element	Required?	Maximum Number In File	Description
<capacity>	Optional	Unbounded	Specifies a capacity Work Manager constraint. A Work Manager constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests. The capacity constraint causes the server to reject requests only when it has reached its capacity. The <capacity> element can take the following child elements: • name—The name of the capacity constraint. • count—The integer thread capacity. See Using Work Managers to Optimize Schedule Work.
<work-manager>	Optional	Unbounded	Specifies the Work Manager that is associated with the application. For more information on the elements you can define within the work-manager element, refer to "work-manager" on page A-35. See Using Work Managers to Optimize Schedule Work for detailed information on Work Managers.
<application-admin-mode-trigger>	Optional	Unbounded	Specifies the number of stuck threads needed to bring the application into administration mode. You can specify the following child elements: • max-stuck-thread-time—The maximum amount of time, in seconds, that a thread should remain stuck. • stuck-thread-count—Number of stuck threads that triggers the stuck thread work manager.
<session-descriptor>	Optional	Unbounded	Specifies a list of configuration parameters for servlet sessions. For more information on the elements you can define within the <session-descriptor> element, refer to "session-descriptor" on page A-37.

ejb

The following table describes the elements you can define within an `ejb` element.

Element	Required?	Maximum Number in File	Description
<entity-cache>	Optional	Unbounded	Zero or more. The entity-cache element is used to define a named application level cache that is used to cache entity EJB instances at runtime. Individual entity beans refer to the application-level cache that they must use, referring to the cache name. There is no restriction on the number of different entity beans that may reference an individual cache. Application-level caching is used by default whenever an entity bean does not specify its own cache in the weblogic-ejb-jar.xml descriptor. Two default caches named ExclusiveCache and MultiVersionCache are used for this purpose. An application may explicitly define these default caches to specify non-default values for their settings. Note that the caching-strategy cannot be changed for the default caches. By default, a cache uses max-beans-in-cache with a value of 1000 to specify its maximum size. Example: <entity-cache> <entity-cache-name>ExclusiveCache</entity-cache-name> <max-cache-size> <megabytes>50</megabytes> </max-cache-size> </entity-cache> For more information on the elements you can define within the entity-cache element, refer to "entity-cache" on page A-12.

Element	Required?	Maximum Number in File	Description
<start-mbds-with- application	Optional	1	Allows you to configure the EJB container to start Message Driven BeanS (MDBS) with the application. If set to true, the container starts MDBS as part of the application. If set to false, the container keeps MDBS in a queue and the server starts them as soon as it has started listening on the ports.

entity-cache

The following table describes the elements you can define within a `entity-cache` element.

Element	Required?	Maximum Number in File	Description
<entity-cache-name>	Required	1	Specifies a unique name for an entity bean cache. The name must be unique within an ear file and may not be the empty string. Example: <entity-cache-name>ExclusiveCache</entity-cache-name>
<max-beans-in-cache>	Optional If you specify this element, you cannot also specify <max-cache-size>.	1	Specifies the maximum number of entity beans that are allowed in the cache. If the limit is reached, beans may be passivated. This mechanism does not take into account the actual amount of memory that different entity beans require. This element can be set to a value of 1 or greater. Default Value: 1000

Element	Required?	Maximum Number in File	Description
<max-cache-size>	Optional If you specify this element, you cannot also specify <max-beans-in-cache>.	1	Used to specify a limit on the size of an entity cache in terms of memory size—expressed either in terms of bytes or megabytes. A bean provider should provide an estimate of the average size of a bean in the weblogic-ejb-jar.xml descriptor if the bean uses a cache that specifies its maximum size using the max-cache-size element. By default, a bean is assumed to have an average size of 100 bytes. For more information on the elements you can define within the ejb element, refer to "max-cache-size" on page A-14.
<max-queries-in-cache>	Optional	1	Specifies the maximum SQL queries that can be present in the entity cache at a given moment.
<caching-strategy>	Optional	1	Specifies the general strategy that the EJB container uses to manage entity bean instances in a particular application level cache. A cache buffers entity bean instances in memory and associates them with their primary key value. The caching-strategy element can only have one of the following values: • Exclusive—Caches a single bean instance in memory for each primary key value. This unique instance is typically locked using the EJB container's exclusive locking when it is in use, so that only one transaction can use the instance at a time. • MultiVersion—Caches multiple bean instances in memory for a given primary key value. Each instance can be used by a different transaction concurrently. Default Value: MultiVersion Example: <caching-strategy>Exclusive</caching-strategy>

max-cache-size

The following table describes the elements you can define within a `max-cache-size` element.

Element	Required?	Maximum Number in File	Description
`<bytes>`	You *must* specify either `<bytes>` or `<megabytes>`	1	The size of an entity cache in terms of memory size, expressed in bytes.
`<megabytes>`	You *must* specify either `<bytes>` or `<megabytes>`	1	The size of an entity cache in terms of memory size, expressed in megabytes.

xml

The following table describes the elements you can define within an `xml` element.

Element	Required?	Maximum Number in File	Description
`<parser-factory>`	Optional	1	The parent element used to specify a particular XML parser or transformer for an enterprise application. For more information on the elements you can define within the `parser-factory` element, refer to "parser-factory" on page A-16.
`<entity-mapping>`	Optional	Unbounded	Zero or More. Specifies the entity mapping. This mapping determines the alternative entity URI for a given public or system ID. The default place to look for this entity URI is the `lib/xml/registry` directory. For more information on the elements you can define within the `entity-mapping` element, refer to "entity-mapping" on page A-17.

parser-factory

The following table describes the elements you can define within a `parser-factory` element.

Element	Required?	Maximum Number in File	Description
`<saxparser-factory>`	Optional	1	Allows you to set the SAXParser Factory for the XML parsing required in this application only. This element determines the factory to be used for SAX style parsing. If you do not specify the `saxparser-factory` element setting, the configured SAXParser Factory style in the Server XML Registry is used. Default Value: Server XML Registry setting
`<document-builder-factory>`	Optional	1	Allows you to set the Document Builder Factory for the XML parsing required in this application only. This element determines the factory to be used for DOM style parsing. If you do not specify the `document-builder-factory` element setting, the configured DOM style in the Server XML Registry is used. Default Value: Server XML Registry setting
`<transformer-factory>`	Optional	1	Allows you to set the Transformer Engine for the style sheet processing required in this application only. If you do not specify a value for this element, the value configured in the Server XML Registry is used. Default value: Server XML Registry setting.

entity-mapping

The following table describes the elements you can define within an `entity-mapping` element.

Element	Required?	Maximum Number in File	Description
`<entity-mapping-name>`	Required	1	Specifies the name for this entity mapping.
`<public-id>`	Optional	1	Specifies the public ID of the mapped entity.
`<system-id>`	Optional	1	Specifies the system ID of the mapped entity.
`<entity-uri>`	Optional	1	Specifies the entity URI for the mapped entity.
`<when-to-cache>`	Optional	1	Legal values are: • cache-on-reference • cache-at-initialization • cache-never The default value is `cache-on-reference`.
`<cache-timeout-interval>`	Optional	1	Specifies the integer value in seconds.

jdbc-connection-pool

Note: The `jdbc-connection-pool` element is deprecated. To define a data source in your Enterprise application, you can package a JDBC module with the application. For more information, see "Packaged JDBC Modules" in *Configuring and Managing WebLogic JDBC*.

The following table describes the elements you can define within a `jdbc-connection-pool` element.

Element	Required?	Maximum Number in File	Description
`<data-source-jndi-name>`	Required	1	Specifies the JNDI name in the application-specific JNDI tree.
`<connection-factory>`	Required	1	Specifies the connection parameters that define overrides for default connection factory settings. • `user-name`—Optional. The `user-name` element is used to override `UserName` in the `JDBCDataSourceFactoryMBean`. • `url`—Optional. The `url` element is used to override `URL` in the `JDBCDataSourceFactoryMBean`. • `driver-class-name`—Optional. The `driver-class-name` element is used to override `DriverName` in the `JDBCDataSourceFactoryMBean`. • `connection-params`—Zero or more. • `parameter+` (`param-value`, `param-name`)—One or more For more information on the elements you can define within the `connection-factory` element, refer to "connection-factory" on page A-19.
`<pool-params>`	Optional	1	Defines parameters that affect the behavior of the pool. For more information on the elements you can define within the `pool-params` element, refer to "pool-params" on page A-20.
`<driver-params>`	Optional	1	Sets behavior on WebLogic Server drivers. For more information on the elements you can define within the `driver-params` element, refer to "driver-params" on page A-28.
`<acl-name>`	Optional	1	DEPRECATED.

connection-factory

The following table describes the elements you can define within a `connection-factory` element.

Element	Required?	Maximum Number in File	Description
`<factory-name>`	Optional	1	Specifies the name of a `JDBCDataSourceFactoryMBean` in the `config.xml` file.

`<connection-properties>`	Optional	1	Specifies the connection properties for the connection factory. Elements that can be defined for the `connection-properties` element are:
			• `user-name`—Optional. Used to override UserName in the JDBCDataSourceFactoryMBean.
			• `password`—Optional. Used to override Password in the JDBCDataSourceFactoryMBean.
			• `url`—Optional. Used to override URL in the JDBCDataSourceFactoryMBean.
			• `driver-class-name`—Optional. Used to override DriverName in the JDBCDataSourceFactoryMBean
			• `connection-params`—Zero or more. Used to set parameters which will be passed to the driver when making a connection. Example:
			`<connection-params>`
			`<parameter>`
			`<description>Desc of param`
			`</description>`
			`<param-name>foo</param-name>`
			`<param-value>xyz</param-value>`
			`</parameter>`
			`</connection-params>`

pool-params

The following table describes the elements you can define within a `pool-params` element.

Element	Required?	Maximum Number in File	Description
<size-params>	Optional	1	Defines parameters that affect the number of connections in the pool. • initial-capacity—Optional. The initial-capacity element defines the number of physical database connections to create when the pool is initialized. The default value is 1. • max-capacity—Optional. The max-capacity element defines the maximum number of physical database connections that this pool can contain. Note that the JDBC Driver may impose further limits on this value. The default value is 1. • capacity-increment—Optional. The capacity-increment element defines the increment by which the pool capacity is expanded. When there are no more available physical connections to service requests, the pool creates this number of additional physical database connections and adds them to the pool. The pool ensures that it does not exceed the maximum number of physical connections as set by max-capacity. The default value is 1. • shrinking-enabled—Optional. The shrinking-enabled element indicates whether or not the pool can shrink back to its initial-capacity when connections are detected to not be in use. • shrink-period-minutes—Optional. The shrink-period-minutes element defines the number of minutes to wait before shrinking a connection pool that has incrementally increased to meet demand. The shrinking-enabled element must be set to true for shrinking to take place. • shrink-frequency-seconds—Optional. • highest-num-waiters—Optional. • highest-num-unavailable—Optional.

Element	Required?	Maximum Number in File	Description
<xa-params>	Optional	1	Defines the parameters for the XA DataSources. • debug-level—Optional. Integer. The debug-level element defines the debugging level for XA operations. The default value is 0. • keep-conn-until-tx-complete-enabled—Optional. Boolean. If you set the keep-conn-until-tx-complete-enabled element to true, the XA connection pool associates the same XA connection with the distributed transaction until the transaction completes. • end-only-once-enabled—Optional. Boolean. If you set the end-only-once-enabled element to true, the XAResource.end() method is only called once for each pending XAResource.start() method. • recover-only-once-enabled—Optional. Boolean. If you set the recover-only-once-enabled element to true, recover is only called one time on a resource. • tx-context-on-close-needed—Optional. Set the tx-context-on-close-needed element to true if the XA driver requires a distributed transaction context when closing various JDBC objects (for example, result sets, statements, connections, and so on). If set to true, the SQL exceptions that are thrown while closing the JDBC objects in no transaction context are swallowed. • new-conn-for-commit-enabled—Optional. Boolean. If you set the new-conn-for-commit-enabled element to true, a dedicated XA connection is used for commit/rollback processing of a particular distributed transaction.

Element	Required?	Maximum Number in File	Description
`<xa-params>` `Continued...`	Optional	1	• `prepared-statement-cache-size`—**Deprecated**. Optional. Use the prepared-statement-cache-size element to set the size of the prepared statement cache. The size of the cache is a number of prepared statements created from a particular connection and stored in the cache for further use. Setting the size of the prepared statement cache to 0 turns it off. **Note:** `Prepared-statement-cache-size` is deprecated. Use `cache-size` in `driver-params/prepared-statement`. See "driver-params" for more information. • `keep-logical-conn-open-on-release`—Optional. Boolean. Set the `keep-logical-conn-open-on-release` element to `true`, to keep the logical JDBC connection open when the physical XA connection is returned to the XA connection pool. The default value is `false`. • `local-transaction-supported`—Optional. Boolean. Set the `local-transaction-supported` to `true` if the XA driver supports SQL with no global transaction; otherwise, set it to `false`. The default value is `false`. • `resource-health-monitoring-enabled`—Optional. Set the `resource-health-monitoring-enabled` element to `true` to enable JTA resource health monitoring for this connection pool.

Element	Required?	Maximum Number in File	Description
<xa-params> Continued...	Optional	1	• xa-set-transaction-timeout—Optional. Used in: xa-params Example: <xa-set-transaction-timeout> true </xa-set-transaction-timeout> • xa-transaction-timeout—Optional. When the xa-set-transaction-timeout value is set to true, the transaction manager invokes setTransactionTimeout on the resource before calling XAResource.start. The Transaction Manager passes the global transaction timeout value. If this attribute is set to a value greater than 0, then this value is used in place of the global transaction timeout. Default value: 0 Used in: xa-params Example: <xa-transaction-timeout> 30 </xa-transaction-timeout> • rollback-localtx-upon-connclose—Optional. When the rollback-localtx-upon-connclose element is true, the connection pool calls rollback() on the connection before putting it back in the pool. Default value: false Used in: xa-params Example: <rollback-localtx-upon-connclose> true </rollback-localtx-upon-connclose>

Element	Required?	Maximum Number in File	Description
`<login-delay-seconds>`	Optional	1	Sets the number of seconds to delay before creating each physical database connection. Some database servers cannot handle multiple requests for connections in rapid succession. This property allows you to build in a small delay to let the database server catch up. This delay occurs both during initial pool creation and during the lifetime of the pool whenever a physical database connection is created.
`<leak-profiling-enabled>`	Optional	1	Enables JDBC connection leak profiling. A connection leak occurs when a connection from the pool is not closed explicitly by calling the `close()` method on that connection. When connection leak profiling is active, the pool stores the stack trace at the time the connection object is allocated from the pool and given to the client. When a connection leak is detected (when the connection object is garbage collected), this stack trace is reported. This element uses extra resources and will likely slowdown connection pool operations, so it is not recommended for production use.

Element	Required?	Maximum Number in File	Description
<connection-check-params>	Optional	1	• Defines whether, when, and how connections in a pool is checked to make sure they are still alive. • table-name—Optional. The table-name element defines a table in the schema that can be queried. • check-on-reserve-enabled—Optional. If the check-on-reserve-enabled element is set to true, then the connection will be tested each time before it is handed out to a user. • check-on-release-enabled—Optional. If the check-on-release-enabled element is set to true, then the connection will be tested each time a user returns a connection to the pool. • refresh-minutes—Optional. If the refresh-minutes element is defined, a trigger is fired periodically (based on the number of minutes specified). This trigger checks each connection in the pool to make sure it is still valid. • check-on-create-enabled—Optional. If set to true, then the connection will be tested when it is created. • connection-reserve-timeout-seconds—Optional. Number of seconds after which the call to reserve a connection from the pool will timeout. • connection-creation-retry-frequency-seconds—Optional. The frequency of retry attempts by the pool to establish connections to the database. • inactive-connection-timeout-seconds—Optional. The number of seconds of inactivity after which reserved connections will forcibly be released back into the pool.

Element	Required?	Maximum Number in File	Description
<connection-check-params> Continued...	Optional	1	• test-frequency-seconds—Optional. The number of seconds between database connection tests. After every test-frequency-seconds interval, unused database connections are tested using table-name. Connections that do not pass the test will be closed and reopened to re-establish a valid physical database connection. If table-name is not set, the test will not be performed. • init-sql—Optional. Specifies a SQL query that automatically runs when a connection is created.
<jdbcxa-debug-level>	Optional	1	This is an internal setting.
<remove-infected-connections-enabled>	Optional	1	Controls whether a connection is removed from the pool when the application asks for the underlying vendor connection object. Enabling this attribute has an impact on performance; it essentially disables the pooling of connections (as connections are removed from the pool and replaced with new connections).

driver-params

The following table describes the elements you can define within a `driver-params` element.

Element	Required?	Maximum Number in File	Description

`<statement>`	Optional	1	Defines the `driver-params` statement. Contains the following optional element: `profiling-enabled`.
			Example:
			`<statement>`
			`<profiling-enabled>true`
			`</profiling-enabled>`
			`</statement>`

Element	Required?	Maximum Number in File	Description
<prepared-statement	Optional	1	Enables the running of JDBC prepared statement cache profiling. When enabled, prepared statement cache profiles are stored in external storage for further analysis. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. The default value is false. • profiling-enabled—Optional. • cache-profiling-threshold—Optional. The cache-profiling-threshold element defines a number of statement requests after which the state of the prepared statement cache is logged. This element minimizes the output volume. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. • cache-size—Optional. The cache-size element returns the size of the prepared statement cache. The size of the cache is a number of prepared statements created from a particular connection and stored in the cache for further use. • parameter-logging-enabled—Optional. During SQL roundtrip profiling it is possible to store values of prepared statement parameters. The parameter-logging-enabled element enables the storing of statement parameters. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. • max-parameter-length—Optional. During SQL roundtrip profiling it is possible to store values of prepared statement parameters. The max-parameter-length element defines maximum length of the string passed as a parameter for JDBC SQL roundtrip profiling. This is a resource-consuming feature, so you should limit the length of data for a parameter to reduce the output volume. • cache-type—Optional.

Element	Required?	Maximum Number in File	Description
<row-prefetch-enabled>	Optional	1	Specifies whether to enable row prefetching between a client and WebLogic Server for each ResultSet. When an external client accesses a database using JDBC through Weblogic Server, row prefetching improves performance by fetching multiple rows from the server to the client in one server access. WebLogic Server ignores this setting and does not use row prefetching when the client and WebLogic Server are in the same JVM
<row-prefetch-size>	Optional	1	Specifies the number of result set rows to prefetch for a client. The optimal value depends on the particulars of the query. In general, increasing this number increases performance, until a particular value is reached. At that point further increases do not result in any significant increase in performance. **Note:** Typically you will not see any increase in performance after 100 rows. The default value should be adequate for most situations. Valid values for this element are between 2 and 65536. The default value is 48.
<stream-chunk-size>	Optional	1	Specifies the data chunk size for streaming data types, which are pulled from WebLogic Server to the client as needed.

security

The following table describes the elements you can define within a `security` element.

Element	Required?	Maximum Number in File	Description
`<realm-name>`	Optional	1	Names a security realm to be used by the application. If none is specified, the system default realm is used

`<security-role-assignment>`	Optional	Unbounded	Declares a mapping between an application-wide security role and one or more WebLogic Server principals.
			Example:
			`<security-role-assignment>`
			`<role-name>`
			`PayrollAdmin`
			`</role-name>`
			`<principal-name>`
			`Tanya`
			`</principal-name>`
			`<principal-name>`
			`Fred`
			`</principal-name>`
			`<principal-name>`
			`system`
			`</principal-name>`
			`</security-role-assignment>`

application-param

The following table describes the elements you can define within a `application-param` element.

Element	Required?	Maximum Number in File	Description
`<description>`	Optional	1	Provides a description of the application parameter.

Element	Required?	Maximum Number in File	Description
<param-name>	Required	1	Defines the name of the application parameter.
<param-value>	Required	1	Defines the value of the application parameter.

classloader-structure

The following table describes the elements you can define within a `classloader-structure` element.

Element	Required?	Maximum Number in File	Description
<module-ref>	Optional	Unbounded	The following list describes the elements you can define within a `module-ref` element: • `module-uri`—Zero or more. Defined within the `module-ref` element.
<classloader-structure>	Optional	Unbounded	Allows for arbitrary nesting of classloader structures for an application. However, for this version of WebLogic Server, the depth is restricted to three levels.

listener

The following table describes the elements you can define within a `listener` element.

Element	Required?	Maximum Number in File	Description
<listener-class>	Required	1	Name of the user's implementation of `ApplicationLifecycleListener`.
<listener-uri>	Optional	1	A JAR file within the EAR that contains the implementation. If you do not specify the `listener-uri`, it is assumed that the class is visible to the application.

startup

The following table describes the elements you can define within a `startup` element.

Warning: Application-scoped startup and shutdown classes have been deprecated in this release of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Chapter 9, "Programming Application Lifecycle Events."

Element	Required?	Maximum Number in File	Description
`<startup-class>`	Required	1	Defines the name of the class to be run when the application is being deployed.
`<startup-uri>`	Optional	1	Defines a JAR file within the EAR that contains the `startup-class`. If `startup-uri` is not defined, then its assumed that the class is visible to the application.

shutdown

The following table describes the elements you can define within a `shutdown` element.

Warning: Application-scoped startup and shutdown classes have been deprecated in this release of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Chapter 9, "Programming Application Lifecycle Events."

Element	Required Optional	Maximum Number in File	Description
`<shutdown-class>`	Required	1	Defines the name of the class to be run when the application is undeployed.
`<shutdown-uri>`	Optional	1	Defines a JAR file within the EAR that contains the `shutdown-class`. If you do not define the `shutdown-uri` element, it is assumed that the class is visible to the application.

work-manager

The following table describes the elements you can define within a `work-manager` element.

See Using Work Managers to Optimize Schedule Work for examples and information on Work Managers.

Element	Required?	Maximum Number in File	Description
`<name>`	Required	1	The name of the Work Manager.
`<response-time-request-class>`	Optional	1	See the description of the `<response-time-request>` element in "weblogic-application" on page A-2 for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<fair-share-request-class>`, `<context-request-class>`, or `<request-class-name>`.
`<fair-share-request-class>`	Optional	1	See the description of the `<fair-share-request>` element in "weblogic-application" on page A-2 for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<response-time-request-class>`, `<context-request-class>`, or `<request-class-name>`.
`<context-request-class>`	Optional	1	See the description of the `<context-request>` element in "weblogic-application" on page A-2 for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<fair-share-request-class>`, `<response-time-request-class>`, or `<request-class-name>`.
`<request-class-name>`	Optional	1	The name of the request class. If you specify this element, you cannot also specify `<fair-share-request-class>`, `<context-request-class>`, or `<response-time-request-class>`.

Element	Required?	Maximum Number in File	Description
<min-threads-constraint>	Optional	1	See the description of the <min-threads-constraint> element in "weblogic-application" on page A-2 for information on this child element of <work-manager>. If you specify this element, you cannot also specify <min-threads-constaint-name>.
<min-threads-constraint-name>	Optional	1	The name of the min-threads constraint. If you specify this element, you cannot also specify <min-threads-constaint>.
<max-threads-constraint>	Optional	1	See the description of the <max-threads-constraint> element in "weblogic-application" on page A-2 for information on this child element of <work-manager>. If you specify this element, you cannot also specify <max-threads-constaint-name>.
<max-threads-constraint-name>	Optional	1	The name of the max-threads constraint. If you specify this element, you cannot also specify <max-threads-constaint>.
<capacity>	Optional	1	See the description of the <capacity> element in "weblogic-application" on page A-2 for information on this child element of <work-manager>. If you specify this element, you cannot also specify <capacity-name>.
<capacity-name>	Optional	1	The name of the thread capacity constraint. If you specify this element, you cannot also specify <capacity>.

Element	Required?	Maximum Number in File	Description
<work-manager-shutdown-trigger>	Optional	1	Used to specify a Stuck Thread Work Manager component that can shut down the Work Manager in response to stuck threads. You can specify the following child elements: • max-stuck-thread-time—The maximum amount of time, in seconds, that a thread should remain stuck. • stuck-thread-count—Number of stuck threads that triggers the stuck thread work manager. If you specify this element, you cannot also specify <ignore-stuck-threads>.
<ignore-stuck-threads>	Optional	1	Specifies whether the Work Manager should ignore stuck threads and never shut down even if threads become stuck. If you specify this element, you cannot also specify <work-manager-shutdown-trigger>.

session-descriptor

The following table describes the elements you can define within a session-descriptor element.

Element	Required?	Maximum Number in File	Description
<timeout-secs>	Optional	1	Specifies the number of seconds after which the session times out. Default value is 3600 seconds.
<invalidation-interval-secs>	Optional	1	Specifies the number of seconds of the invalidation trigger interval. Default value is 60 seconds.

Element	Required?	Maximum Number in File	Description
<debug-enabled>	Optional	1	Specifies whether debugging is enabled for HTTP sessions. Default value is false.
<id-length>	Optional	1	Specifies the length of the session ID. Default value is 52.
<tracking-enabled>	Optional	1	Specifies whether session tracking is enabled between HTTP requests. Default value is true.
<cache-size>	Optional	1	Specifies the cache size for JDBC and file persistent sessions. Default value is 1028.
<max-in-memory-sessions>	Optional	1	Specifies the maximum sessions limit for memory/replicated sessions. Default value is -1, or unlimited.
<cookies-enabled>	Optional	1	Specifies the Web application container should set cookies in the response. Default value is true.
<cookie-name>	Optional	1	Specifies the name of the cookie that tracks sessions. Default name is JSESSIONID.
<cookie-path>	Optional	1	Specifies the session tracking cookie path. Default value is /.
<cookie-domain>	Optional	1	Specifies the session tracking cookie domain. Default value is null.
<cookie-comment>	Optional	1	Specifies the session tracking cookie comment. Default value is null.

Element	Required?	Maximum Number in File	Description
<cookie-secure>	Optional	1	Specifies whether the session tracking cookie is marked secure. Default value is false.
<cookie-max-age-secs>	Optional	1	Specifies that maximum age of the session tracking cookie. Default value is -1, or unlimited.
<persistent-store-type>	Optional	1	Specifies the type of storage for session persistence. You can specify the following values: • memory—Default value. • replicated—Requires clustering. • replicated_if_clustered—Defaults to memory in non-clustered case. • file • jdbc • cookie
<persistent-store-cookie-name>	Optional	1	Specifies the name of the cookie that holds the attribute name and values when using cookie-based session persistence. Default value is WLCOOKIE.
<persistent-store-dir>	Optional	1	Specifies the name of the directory when using file-based session persistence. The directory is relative to the temporary directory defined for the Web application. Default value is session_db.
<persistent-store-pool>	Optional	1	Specifies the name of the JDBC connection pool when using jdbc-based session persistence.
<persistent-store-table>	Optional	1	Specifies the name of the database table when using jdbc-based session persistence. Default value is wl_servlet_sessions.

Element	Required?	Maximum Number in File	Description
<jdbc-column-name-max-inactive-interval>	Optional	1	Alternative name for the wl_max_inactive_interval column name when using jdbc-based session persistence. Required for certain databases that do not support long column names
<jdbc-connection-timeout-secs>	Optional	1	DEPRECATED
<url-rewriting-enabled>	Optional	1	Specifies whether URL rewriting is enabled. Default value is true.
<http-proxy-caching-of-cookies>	Optional	1	Specifies whether WebLogic Server adds the following HTTP header to the response: Cache-control: no-cache=set-cookie This header specifies that proxy caches should not cache the cookies. Default value is true, which means that the header is NOT added. Set this element to false if you want the header added to the response.
<encode-session-id-in-query-params>	Optional	1	Specifies whether WebLogic Server should encode the session ID in the path parameters. Default value is false.
<monitoring-attribute-name>	Optional	1	Used to tag runtime information for different sessions. For example, set this element to username if you have a username attribute that is guaranteed to be unique.
<sharing-enabled>	Optional	1	Specifies whether HTTP sessions are shared across multiple Web applications. Default value is false.

library

The following table describes the elements you can define within a library element.

See Chapter 8, "Creating Shared J2EE Libraries and Optional Packages," for additional information and examples.

Element	Required?	Maximum Number in File	Description
`<library-name>`	Required	1	Specifies the name of the referenced shared J2EE library.
`<specification-version>`	Optional	1	Specifies the minimum specification-version required.
`<implementation-version>`	Optional	1	Specifies the minimum implementation-version required.
`<exact-match>`	Optional	1	Specifies whether there must be an exact match between the specification and implementation version that is specified and that of the referenced library. Default value is `false`.
`<context-root>`	Optional	1	Specifies the context-root of the references Web Applications shared J2EE library.

weblogic-application.xml Schema

See http://www.bea.com/ns/weblogic/90/weblogic-application.xsd for the XML Schema of the `weblogic-application.xml` deployment descriptor file.

application.xml Schema

For more information about `application.xml` deployment descriptor elements, see the J2EE 1.4 schema available at http://java.sun.com/xml/ns/j2ee/application_1_4.xsd.

APPENDIX B

wldeploy Ant Task Reference

The following sections describe tools for deploying applications and standalone modules to WebLogic Server:

Overview of the wldeploy Ant Task

The `wldeploy` Ant task enables you to perform `weblogic.Deployer` functions using attributes specified in an Ant XML file. You can use `wldeploy` along with other WebLogic Server Ant tasks to create a single Ant build script that:

- Builds your application from source, using `wlcompile`, `appc`, and the Web Services Ant tasks.
- Creates, starts, and configures a new WebLogic Server domain, using the `wlserver` and `wlconfig` Ant tasks.
- Deploys a compiled application to the newly-created domain, using the `wldeploy` Ant task.

See "Using Ant Tasks to Configure and Use a WebLogic Server Domain" on page 2-1 for more information about `wlserver` and `wlconfig`. See "Building Applications in a Split Development Directory" on page 4-1 for information about `wlcompile`.

Basic Steps for Using wldeploy

To use the `wldeploy` Ant task:

1. Set your environment.

   On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory `WL_HOME\server\bin`, where `WL_HOME` is the top-level directory of your WebLogic Server installation.

   On UNIX, execute the `setWLSEnv.sh` command, located in the directory `WL_HOME/server/bin`, where `WL_HOME` is the top-level directory of your WebLogic Server installation.

2. In the staging directory, create the Ant build file (`build.xml` by default). If you want to use an Ant installation that is different from the one installed with WebLogic Server, start by defining the `wldeploy` Ant task definition:

   ```
   <taskdef name="wldeploy"
   classname="weblogic.ant.taskdefs.management.WLDeploy"/>
   ```

3. If necessary, add task definitions and calls to the `wlserver` and `wlconfig` tasks in the build script to create and start a new WebLogic Server domain. See "Using Ant Tasks to Configure and Use a WebLogic Server Domain" on page 2-1 for information about `wlserver` and `wlconfig`.

4. Add a call to `wldeploy` to deploy your application to one or more WebLogic Server instances or clusters. See "Sample build.xml Files for wldeploy" on page B-2 and "wldeploy Ant Task Attribute Reference" on page B-4.

5. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   ```
   prompt> ant
   ```

Sample build.xml Files for wldeploy

The following example shows a `wldeploy` target that deploys an application to a single WebLogic Server instance:

```
<target name="deploy">
  <wldeploy
    action="deploy" verbose="true" debug="true"
    name="DeployExample" source="output/redeployEAR"
    user="weblogic" password="weblogic"
    adminurl="t3://localhost:7001" targets="myserver" />
</target>
```

The following example shows a corresponding task to undeploy the application; the example shows that when you undeploy or redeploy an application, you do not specify the source archive file or exploded directory, but rather, just its deployed name.:

```
<target name="undeploy">
  <wldeploy
    action="undeploy" verbose="true" debug="true"
    name="DeployExample"
    user="weblogic" password="weblogic"
    adminurl="t3://localhost:7001" targets="myserver"
    failonerror="false" />
</target>
```

The following example shows how to perform a partial redeploy of the application; in this case, just a single WAR file in the application is redeployed:

```
<target name="redeploy_partial">
  <wldeploy
    action="redeploy" verbose="true"
    name="DeployExample"
    user="weblogic" password="weblogic"
    adminurl="t3://localhost:7001" targets="myserver"
    deltaFiles="examples/general/redeploy/SimpleImpl.war" />
</target>
```

The following example uses the nested `<files>` child element of `wldeploy` to specify a particular file in the application that should be undeployed:

```
<target name="undeploy_partial">
  <wldeploy
    action="undeploy" verbose="true" debug="true"
    name="DeployExample"
    user="weblogic" password="weblogic"
```

```
      adminurl="t3://localhost:7001" targets="myserver"
      failonerror="false">
      <files
         dir="${current-dir}/output/redeployEAR/examples/general/redeploy"
         includes="SimpleImpl.jsp" />
    </wldeploy>
  </target>
```

The following example shows how to deploy a J2EE library called `myLibrary` whose source files are located in the `output/myLibrary` directory:

```
  <target name="deploy">
    <wldeploy action="deploy" name="myLibrary"
      source="output/myLibrary" library="true"
      user="weblogic" password="weblogic"
      verbose="true" adminurl="t3://localhost:7001"
      targets="myserver" />
  </target>
```

wldeploy Ant Task Attribute Reference

The following sections describe the attributes and child element `<files>` of the `wldeploy` Ant task.

Main Attributes

The following table describes the main attributes of the `wldeploy` Ant task.

These attributes mirror some of the arguments of the `weblogic.Deployer` command. BEA provides an Ant task version of the `weblogic.Deployer` command so that developers can easily deploy and test their applications as part of the iterative development process. Typically, however, administrators use the `weblogic.Deployer` command, and not the `wldeploy` Ant task, to deploy applications in a production environment. For that reason, see the weblogic.Deployer Command-Line Reference in *Deploying Applications to WebLogic Server* for

the full and complete definition of the attributes of the `wldeploy` Ant task. The table below is provided just as a quick summary.

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
action	The deployment action to perform. Valid values are `deploy`, `cancel`, `undeploy`, `redeploy`, `distribute`, `start`, and `stop`.	String
adminmode	Specifies that the deployment action puts the application into Administration mode. Administration mode restricts access to an application to a configured Administration channel. Valid values for this attribute are `true` and `false`. Default value is `false`, which means that by default the application is deployed in production mode so that all clients can access it immediately.	Boolean
adminurl	The URL of the Administration Server. The format of the value of this attribute is *`protocol`*`://`*`host`*`:`*`port`*, where *`protocol`* is either `http` or `t3`, *`host`* is the host on which the Administration Server is running, and *`port`* is the port which the Administration Server is listening. **Note:** In order to use the HTTP protocol, you must enable the http tunnelling option in the Administration Console.	String
altappdd	Specifies the name of an alternate J2EE deployment descriptor (`application.xml`) to use for deployment. If you do not specify this attribute, and you are deploying an Enterprise application, the default deployment descriptor is called `application.xml` and is located in the META-INF subdirectory of the main application directory or archive (specified by the `source` attribute.)	String
altwlsappdd	Specifies the name of an alternate WebLogic Server deployment descriptor (`weblogic-application.xml`) to use for deployment. If you do not specify this attribute, and you are deploying an Enterprise application, the default deployment descriptor is called `weblogic-application.xml` and is located in the META-INF subdirectory of the main application directory or archive (specified by the `source` attribute.)	String

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
appversion	The version identifier of the deployed application.	String
debug	Enable `wldeploy` debugging messages.	Boolean
deltaFiles	Specifies a comma- or space-separated list of files, relative to the root directory of the application, which are to be redeployed. Use this attribute only in conjunction with `action="redeploy"` to perform a partial redeploy of an application.	String
external_stage	Specifies whether the deployment uses `external_stage` deployment mode. In this mode, the Ant task does not copy the deployment files to target servers; instead, you must ensure that deployment files have been copied to the correct subdirectory in the target servers' staging directories. You can specify only one of the following attributes: `stage`, `nostage`, or `external_stage`. If none is specified, the default deployment mode to Managed Servers is `stage`; the default mode to the Administration Server and in single-server cases is `nostage`. See Controlling Deployment File Copying with Staging Modes.	Boolean
failonerror	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. Valid values for this attribute are `true` and `false`. Default value is `true`.	Boolean
graceful	Stops the application after existing HTTP clients have completed their work. You can use this attribute *only* when stopping or undeploying an application, or in other words, you must also specify either the `action="stop"` or `action="undeploy"` attributes. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
id	Identification used for obtaining status or cancelling the deployment. You assign a unique ID to an application when you deploy it, and then subsequently use the ID when redeploying, undeploying, stopping, and so on. If you do not specify this attribute, the Ant task assigns a unique ID to the application.	String

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
ignoresessions	This option immediately places the application into Administration mode without waiting for current HTTP sessions to complete. You can use this attribute *only* when stopping or undeploying an application, or in other words, you must also specify either the `action="stop"` or `action="undeploy"` attributes. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
libImplVer	Specifies the implementation version of a J2EE library or optional package. This attribute can be used only if the library or package does not include a implementation version in its manifest file. You can specify this attribute only in combination with the `library` attribute. See "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.	String
library	Identifies the deployment as a shared J2EE library or optional package. You must specify the `library` attribute when deploying or distributing any J2EE library or optional package. Valid values for this attribute are `true` and `false`. Default value is `false`. See "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.	Boolean
libSpecVer	Provides the specification version of a J2EE library or optional package. This attribute can be used only if the library or package does not include a specification version in its manifest file. You can specify this attribute only in combination with the `library` attribute. See "Creating Shared J2EE Libraries and Optional Packages" on page 8-1.	String
name	The deployment name for the deployed application. If you do not specify this attribute, WebLogic Server assigns a deployment name to the application, based on its archive file or exploded directory.	String

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
nostage	Specifies whether the deployment uses nostage deployment mode. In this mode, the Ant task does not copy the deployment files to target servers, but leaves them in a fixed location, specified by the `source` attribute. Target servers access the same copy of the deployment files. You can specify only one of the following attributes: `stage`, `nostage`, or `external_stage`. If none is specified, the default deployment mode to Managed Servers is `stage`; the default mode to the Administration Server and in single-server cases is `nostage`. See Controlling Deployment File Copying with Staging Modes.	Boolean
nowait	Specifies whether `wldeploy` returns immediately after making a deployment call (by deploying as a background task).	Boolean
password	The administrative password. **To avoid having the plain text password appear in the build file or in process utilities such as** `ps`, first store a valid username and encrypted password in a configuration file using the `weblogic.Admin STOREUSERCONFIG` command. Then omit both the `username` and `password` attributes in your Ant build file. When the attributes are omitted, `wldeploy` attempts to login using values obtained from the default configuration file. If you want to obtain a username and password from a non-default configuration file and key file, use the `userconfigfile` and `userkeyfile` attributes with `wldeploy`. See STOREUSERCONFIG in the *weblogic.Admin Command-Line Reference* for more information on storing and encrypting passwords.	String
plan	Specifies a deployment plan to use when deploying the application or module. By default, `wldeploy` does not use an available deployment plan, even if you are deploying from an application root directory that contains a plan.	String
planversion	The version identifier of the deployment plan.	String

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
remote	Specifies whether the server is located on a different machine. This affects how filenames are transmitted. Valid values for this attribute are `true` and `false`. Default value is `false`, which means that the Ant task assumes that all source paths are valid paths on the local machine.	Boolean
retiretimeout	Specifies the number of seconds before WebLogic Server undeploys the currently-running version of this application or module so that clients can start using the new version. It is assumed, when you specify this attribute, that you are starting, deploying, or redeploying a new version of an already-running application. See Updating Applications in a Production Environment.	int
securityModel	Specifies the security model to use for this deployment. Possible security models are: • Deployment descriptors only • Customize roles • Customize roles and policies • Security realm configuration (advanced model) Valid actual values for this attribute are `DDOnly`, `CustomRoles`, `CustomRolesAndPolicy`, or `Advanced`. See Options for Securing EJB and Web Application Resources for more information on these security models	String
source	The archive file or exploded directory to deploy.	File
stage	Specifies whether the deployment uses stage deployment mode. In this mode, the Ant task copies deployment files to target servers' staging directories. You can specify only one of the following attributes: `stage`, `nostage`, or `external_stage`. If none is specified, the default deployment mode to Managed Servers is `stage`; the default mode to the Administration Server and in single-server cases is `nostage`. See Controlling Deployment File Copying with Staging Modes.	Boolean

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
submoduletargets	Specifies JMS server targets for resources defined within a JMS application module. The value of this attribute is a comma-separated list of JMS server names. See the Using Sub-Module Targeting with JMS Application Modules.	String
targets	The list of target servers to which the application is deployed. The value of this attribute is a comma-separated list of the target servers, clusters, or virtual hosts. If you do not specify a target list when deploying an application, the target defaults to the Administration Server instance.	String
timeout	The maximum time to wait for a deployment to succeed.	int
upload	Specifies whether the source file(s) are copied to the Administration Server's upload directory prior to deployment. Use this attribute when you are on a remote machine and you cannot copy the deployment files to the Administration Server by other means. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
usenonexclusivelock	Specifies that the deployment action (deploy, redeploy, stop, and so on) uses the existing lock on the domain that has already been acquired by the same user performing the action. This attribute is particularly useful when the user is using multiple deployment tools (Ant task, command line, Administration console, and so on) simultaneously and one of the tools has already acquired a lock on the domain. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
user	The administrative username.	String
userconfigfile	Specifies the location of a user configuration file to use for obtaining the administrative username and password. Use this option, instead of the `user` and `password` attributes, in your build file when you do not want to have the plain text password shown in-line or in process-level utilities such as `ps`. Before specifying the `userconfigfile` attribute, you must first generate the file using the `weblogic.Admin STOREUSERCONFIG` command as described in STOREUSERCONFIG in the *weblogic.Admin Command-Line Reference*.	String

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
userkeyfile	Specifies the location of a user key file to use for encrypting and decrypting the username and password information stored in a user configuration file (the `userconfigfile` attribute). Before specifying the `userkeyfile` attribute, you must first generate the key file using the `weblogic.Admin STOREUSERCONFIG` command as described in STOREUSERCONFIG in the *weblogic.Admin Command-Line Reference*.	String
verbose	Specifies whether `wldeploy` displays verbose output messages.	Boolean

Nested <files> Child Element

The `wldeploy` Ant task also includes the `<files>` child element that can be nested to specify a list of files on which to perform a deployment action (for example, a list of JSPs to undeploy.)

Warning: Use of `<files>` to redeploy a list of files in an application has been deprecated in this release. Instead, use the `deltaFiles` attribute of wldeploy.

The `<files>` element works the same as the standard `<fileset>` Ant task (except for the difference in actual task name). Therefore, see the Apache Ant Web site for detailed reference information about the attributes you can specify for the `<files>` element.

APPENDIX C

Spring Applications Reference

The following sections describe developing and managing Spring Framework-based applications for WebLogic Server. In most cases, the information in these sections is described from the perspective of creating MedRec-Spring.

About Spring on WebLogic Server

To demonstrate the ways in which Spring can take advantage of WebLogic Server's enterprise features, BEA redesigned the Avitek Medical Records sample application (MedRec) to replace core J2EE components with Spring components. For additional information on MedRec architecture and its redesign see the article "Spring Integration with WebLogic Server" at http://dev2dev.bea.com/pub/a/2005/09/spring_integration_weblogic_server.html.

The following sections describe key steps that BEA performed when redesigning MedRec. You can use this information if you want to redesign your own J2EE-based WebLogic Server applications to use Spring components. You can also leverage this information if you want to create a new application, based on Spring components, for WebLogic Server.

It is assumed that you are familiar with J2EE concepts, WebLogic Server 9.0, and the Spring Framework. For information on WebLogic Server 9.0, see BEA WebLogic Server 9.0

Documentation. For information on the Spring Framework, see http://www.springframework.org/.

Redesigning a J2EE-Based Application to a Spring-Based Application

To transform a J2EE-based application to a Spring-based application, you perform the following steps as desired:

1. Configure Spring Inversion of Control.
2. Enable the Spring Web Services Client Service. Spring offers a JAX-RPC factory which produces a proxy for Web Services.
3. Make JMS Services Available to the Application at Runtime.
4. Configure JMX: Expose the WebLogic Server Runtime MBean Server Connection to Spring.
5. Configure Spring JDBC to Communicate With the Connection Pool.
6. Use the Spring Transaction Abstraction Layer for Transaction Management.
7. Make Use of WebLogic Server Clustering and Clustered Spring Remoting.

The following sections describe the details of redesigning a J2EE-based application to a Spring-based application. Where appropriate, these sections include sample code. In most cases the sample code is from MedRec-Spring.

Configure Spring Inversion of Control

In Spring, references to other beans (injected properties) are configured via a Spring configuration XML file, `applicationContext-web.xml`.

In MedRec-Spring, BEA replaced stateless session EJBs with POJOs in the Spring configuration file `src\medrecEar\web\WEB-INF\applicationContext-web.xml` as follows:

```
<bean name="/patient/record"
    class="com.bea.medrec.web.patient.actions.ViewRecordAction">
    <property name="medRecClientServiceFacade">
      <ref bean="medRecClientServiceFacade"/>
    </property>
  </bean>
```

Then, in the application code, BEA defined setter methods for the corresponding bean. For example:

```
protected MedRecClientServiceFacade medRecClientServiceFacade;

  public void setMedRecClientServiceFacade(

      MedRecClientServiceFacade pMedRecClientServiceFacade){

    this.medRecClientServiceFacade = pMedRecClientServiceFacade;

  }
```

Enable the Spring Web Services Client Service

To use Spring's JAX-RPC factory which produces a proxy for Web Services, you configure the Spring `JaxRpcPortProxyFactoryBean` by implementing code such as the following; in MedRec-Spring, BEA implemented this code in the Spring configuration file `src\physicianEar\APP-INF\classes\applicationContext-phys-service.xml`.

```
<!-- reliable asynchronous web service for sending new medical records to
medrec -->
<bean id="reliableClientWebServicesPortType"
class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean"
lazy-init="true">
<property name="wsdlDocumentUrl"
value="http://${WS_HOST}:${WS_PORT}/ws_phys/PhysicianWebServices?WSDL"/>
<property name="portName" value="PhysicianWebServicesPort"/>
<property name="jaxRpcService">
<ref bean="generatedReliableService"/>
</property>
<property name="serviceInterface"
value="com.bea.physician.webservices.client.PhysicianWebServicesPortType"/
>
<property name="username" value="medrec_webservice_user"/>
<property name="password" value="weblogic"/>
<property name="customProperties">
<props>
<prop key="weblogic.wsee.complex">true</prop>
</props>
</property>
</bean>
```

```
<> <!-- allows the jaxRpcService class to execute its constructor which
loads in type mappings -->
<bean id="generatedReliableService"
class="com.bea.physician.webservices.client.PhysicianWebServices_Impl">
</bean>
```

In this code example, note that:

- The `serviceInterface` represents Web Services operations.
- The `customProperties` property allows for custom WebLogic Server Web Service stub properties.
- The `jaxRpcService` value is set to WebLogic Server's generated JAX-RPC implementation service.

Make JMS Services Available to the Application at Runtime

In Spring, you must configure JMS services so that they are provided to the application during runtime. You can do this via a Spring Bean that represents a messaging destination. In Med-Rec Spring, BEA made JMS services available to the application at runtime by implementing the following code in the Spring configuration file `src\medrecEar\APP-INF\classes\applicationContext-jms.xml`.

```
<bean id="uploadQueue"
class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"
      value="com.bea.medrec.messagging.MedicalRecordUploadQueue"/>
  </bean>

  <bean id="jmsConnFactory"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"
      value="com.bea.medrec.messagging.MedRecQueueConnectionFactory"/>
  </bean>
```

```
  <bean id="uploadJmsTemplate"
class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory">
      <ref bean="jmsConnFactory"/>
    </property>
    <property name="defaultDestination">
      <ref bean="uploadQueue"/>
    </property>
  </bean>
```

Configure JMX: Expose the WebLogic Server Runtime MBean Server Connection to Spring

You can expose WebLogic Server's MBean Server to Spring through Spring's `MBeanServerConnectionFactoryBean`, which is a convenience factory that produces an `MBeanServerConnection` that is established and cached during application deployment and can later be operated on by referencing beans. The `MBeanServerConnectionFactoryBean` can be configured to return the WebLogic Server Runtime MBean Server, and to obtain a connection to the WebLogic Server Domain Runtime MBean Server and the WebLogic Server Edit MBean Server.

Note: Because the WebLogic Server Domain Runtime MBean Server is not active during deployment, you must configure the `MBeanServerConnectionFactoryBean` to use Spring's lazy instantiation. Lazy instantiation fetches the Spring Bean when it is invoked.

Exposing the WebLogic Server Runtime MBean Server Connection to Spring is demonstrated in the following code example, which, in MedRec-Spring, BEA implemented in the Spring configuration file `medrecEar/APP-INF/classes/applicationContext-jmx.xml`.

```
<> <!-- expose weblogic server's runtime mbeanserver connection -->
<bean id="runtimeMbeanServerConnection"
class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
<property name="serviceUrl"
value="service:jmx:t3://${WS_HOST}:${WS_PORT}/jndi/weblogic.management.mbe
anservers.runtime"/>
<property name="environment">
<props>
```

```
<prop key="java.naming.security.principal">${WS_USERNAME}</prop>
<prop key="java.naming.security.credentials">${WS_USERNAME}</prop>
<prop
key="jmx.remote.protocol.provider.pkgs">weblogic.management.remote</prop>
</props>
</property>
</bean>
```

Configure Spring JDBC to Communicate With the Connection Pool

In MedRec-Spring, BEA used a data source that references a JDBC connection pool that is managed by WebLogic Server and also employed Spring's `JdbcDaoSupport` class. For information on `JdbcDaoSupport`, see the Spring documentation.

For an example of the way in which BEA implemented JDBC, see the MedRec-Spring class

`src\medrecEar\dao\com\bea\medrec\dao\jdbc\JdbcPatientDao.java`

See also the following code examples, which, for MedRec-Spring, BEA implemented in the Spring configuration files `src\medrecEar\APP-INF\classes\applicationContext-db.xml` and `src\medrecEar\APP-INF\classes\applicationContext-jdbc.xml`, respectively.

`applicationContext-db.xml` code example:

```
  <!-- datasource pool -->
  <bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MedRecGlobalDataSourceXA"/>
  </bean>
```

`applicationContext-jdbc.xml` code example:

```
  <bean id="patientDao"
    class="com.bea.medrec.dao.jdbc.JdbcPointBasePatientDao"
    autowire="byType"/>
```

Additionally, in MedRec-Spring, BEA replaced entity EJBs with POJOs and made use of Spring JDBC for persistence. For an example, see the MedRec-Spring class `\src\medrecEar\core\com\bea\medrec\domain\Address.java`

Use the Spring Transaction Abstraction Layer for Transaction Management

Spring supports distributed transactions through WebLogic Server's JTA implementation. You can also configure the Spring transaction manager to delegate responsibility to the WebLogic Server JTA transaction manager. This is accomplished via Spring's `WebLogicJtaTransactionManager` class. BEA used this approach with MedRec-Spring in order to exactly mirror transaction management in the original version of MedRec.

To use the Spring transaction abstraction layer for transaction management and delegate responsibility to the WebLogic Server JTA transaction manager, you implement code such as the following, which BEA implemented in the Spring configuration files `src\medrecEar\APP-INF\classes\applicationContext-tx.xml` and `src\medrecEar\APP-INF\classes\applicationContext-service.xml`, respectively.

`applicationContext-tx.xml` code example:

```
<!-- spring's transaction manager delegates to WebLogic Server's transaction
manager -->
<bean id="transactionManager"
class="org.springframework.transaction.jta.WebLogicJtaTransactionManager">
<property name="transactionManagerName"
value="javax.transaction.TransactionManager"/>
</bean>
```

`applicationContext-service.xml` code example:

```
<!-- base transaction proxy for which medrec spring beans inherit-->
< bean id="baseTransactionProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactory
Bean"
abstract="true">
<property name="transactionManager" ref="transactionManager"/>
<property name="transactionAttributes">
<props>
<prop key="activate*">PROPAGATION_REQUIRED</prop>
<prop key="create*">PROPAGATION_REQUIRED</prop>
```

```
<prop key="compose*">PROPAGATION_REQUIRED</prop>
<prop key="deny*">PROPAGATION_REQUIRED</prop>
<prop key="getRecord*">PROPAGATION_REQUIRED,readOnly</prop>
<prop key="getPatient*">PROPAGATION_REQUIRED,readOnly</prop>
<prop key="getLog*">PROPAGATION_NOT_SUPPORTED</prop>
<prop key="process*">PROPAGATION_REQUIRED</prop>
<prop key="save*">PROPAGATION_REQUIRED</prop>
<prop key="send*">PROPAGATION_REQUIRED</prop>
</props>
</property>
< /bean>

<!-- single point of service for all medrec clients -->
<bean id="medRecClientServiceFacade"
parent="baseTransactionProxy">
<property name="target">
<bean class="com.bea.medrec.service.MedRecClientServiceFacadeImpl">
<property name="adminService">
<ref bean="adminService"/>
</property>
<property name="patientService">
<ref bean="patientService"/>
</property>
<property name="recordService">
<ref bean="recordService"/>
</property>
<property name="recordXmlProcessorService">
<ref bean="recordXmlProcessorService"/>
</property>
</bean>
</property>
</bean>
```

The `transactionAttributes` you specify define the way in which Spring begins and ends transactions. Because MedRec-Spring delegates transaction management to WebLogic JTA, management tasks such as transaction suspension and rollback are handled as specified by WebLogic's transaction manager.

For more information on `WebLogicJtaTransactionManager`, see "Implementing Transaction Suspension in Spring" at http://dev2dev.bea.com/pub/a/2005/07/spring_transactions.html.

Make Use of WebLogic Server Clustering

Spring applications can take advantage of WebLogic Server's clustering features. Because most Spring applications are packaged as Web applications (.war files), you need do not need to do anything special in order to take advantage of WebLogic Server clusters; all you need to do is deploy your Spring application to the servers in a WebLogic Server cluster.

Clustered Spring Remoting

The certification of Spring 1.2.5 on WebLogic Server 9.0 extends the Spring `JndiRmiProxyFactoryBean` and its associated service exporter so that it supports proxying with any J2EE RMI implementation. To use the extension to the `JndiRmiProxyFactoryBean` and its exporter:

1. Configure client support by implementing code such as the following:

```
<bean id="proProxy"
class="org.springframework.remoting.rmi.JndiRmiProxyFactoryBean">
<property name="jndiName" value="t3://${serverName}:${rmiPort}/order"/>
</property>
<property name="jndiEnvironment">
<props>
<prop key="java.naming.factory.url.pkgs">weblogic.jndi.factories</prop>
</props>
</property>
<property name="serviceInterface"
value="org.springframework.samples.jpetstore.domain.logic.OrderService"
/>
</bean>
```

2. Configure the service exporter by implementing code such as the following:

```
<bean id="order-pro"
class="org.springframework.remoting.rmi.JndiRmiServiceExporter">
<property name="service" ref="petStore"/>
<property name="serviceInterface"
value="org.springframework.samples.jpetstore.domain.logic.OrderService"
/>
<property name="jndiName" value="order"/>
</bean>
```

Spring Extension to the WebLogic Administration Console

You can use a Spring extension to the WebLogic Server Administration Console to monitor and manage Spring Beans, attributes, and operations that are defined in your application.

Installing the Spring Extension to the WebLogic Administration Console

To install the Spring extension to the WebLogic Administration Console, perform the following steps:

1. Copy the `spring-ext-server.jar` file to your `yourdomain/console-ext` directory.
2. Copy the `spring-ext-client.jar` file to your application's `WEB-INF/lib` directory.
3. Restart WebLogic Server.

Exposing Spring Beans Through the WebLogic Administration Console

In order to be able to access Spring Beans that are not MBeans through the Web Logic Administration Console, you must configure an MBeanExporter in the `applicationContext.xml` file and specify which beans to expose via the assembler. Make sure that the `applicationName` property is the deployed name of your application.

Support for Spring on WebLogic Server

For information on how BEA supports this release of WebLogic Server and the Spring Framework from Interface21, see Supported Configurations for Products with Spring Framework.