BEA WebLogic Server®

Developing Applications with WebLogic Server

Version 9.0 BETA
Revised: December 15, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Overview of WebLogic Server Application Development

2. Using Ant Tasks to Configure and Use a WebLogic Server Domain

3. Creating a Split Development Directory Environment

4. Building Applications in a Split Development Directory

5. Deploying and Packaging from a Split Development Directory

6. Understanding WebLogic Server Application Classloading

7. Developing Applications for Production Redeployment

8. Exporting an Application for Deployment to New Environments

9. Creating Shared J2EE Libraries and Optional Packages

10.Programming Application Lifecycle Events

11.Programming JavaMail with WebLogic Server

12.Threading and Clustering Topics

A. Enterprise Application Deployment Descriptor Elements

B. weblogic.Configure Command Line Reference

C. wldeploy Ant Task 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 1.3 technologies (`http://java.sun.com/j2ee/sdk_1.3/index.html`). 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, 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 1.3 applications on WebLogic Server versions 7.x 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 1.3 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 4-5.

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 4-5.

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`
	WebLogic	`weblogic.xml` See "weblogic.xml Deployment Descriptor Elements" in *Developing Web Applications for WebLogic Server*.
Enterprise Bean	J2EE	`ejb-jar.xml` See the Sun Microsystems EJB 2.0 DTD.

	WebLogic	`weblogic-ejb-jar.xml`
		See "The weblogic-ejb-jar.xml Deployment Descriptor" in *Programming WebLogic Enterprise JavaBeans*.
		`weblogic-cmp-rdbms-jar.xml`
		See "The weblogic-cmp-rdbms-jar.xml Deployment Descriptor" in *Programming WebLogic Enterprise JavaBeans*.

Resource Adapter	J2EE	`ra.xml` See the Sun Microsystems Connector 1.0 DTD.
	WebLogic	`weblogic-ra.xml` See "weblogic-ra.xml Schema" in *Programming WebLogic Server Resource Adapters*.
Enterprise Application	J2EE	`application.xml` See "application.xml Deployment Descriptor Elements" on page A-2.
	WebLogic	`weblogic-application.xml` See "weblogic-application.xml Deployment Descriptor Elements" on page A-2.

Table 1-1 J2EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Client Application	J2EE	`application-client.xml` See *Programming Client Applications to WebLogic Server*.
	WebLogic	`weblogic-appclient.xml` See *Programming Client Applications to WebLogic Server*.

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 JavaBeans (version 2.0).

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, EJB 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.
- `java weblogic.marathon.ddinit.EJBInit`—automatically generates the deployment descriptors for Enterprise JavaBeans 2.0. If `ejb-jar.xml` exists, `DDInit` uses its deployment information to generate `weblogic-ejb-jar.xml`.

For an example of `DDInit`, assume that you have created a directory called `c:\stage` that contains the `WEB-INF` directory, 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:

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.

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. 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 the `ANT_OPTS` environment variable. For example:

```
setenv ANT_OPTS=-Xmx128m
```

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

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.

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 or XMLSpy (bundled as part of the WebLogic Server package). 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 following sections describe how to use the `wlserver` Ant task to connect to an existing WebLogic Server domain.

What the wlserver Ant Task Does

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.

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, reboot, or connect to.	String	No
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
timeout	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.	long	No
productionmodeenabled	Specifies whether a server instance boots in development mode or in production mode.	boolean	No
host	The DNS name or IP address on which the server instance is listening.	String	No
port	The TCP port number on which the server instance is listening.	int	No

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
generateconfig	Specifies whether or not `wlserver` creates a new domain for the specified server. This attribute is false by default.	boolean	No
action	Specifies the action `wlserver` performs: `startup`, `shutdown`, `reboot`, or `connect`. The `shutdown` action can be used with the optional `forceshutdown` attribute perform a forced shutdown.	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
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"/>`	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.

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

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.	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 2-11.	String	Yes
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.	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
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. `query` can be used with nested `set` elements or a nested `delete` element to perform set or delete operations on all 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

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)

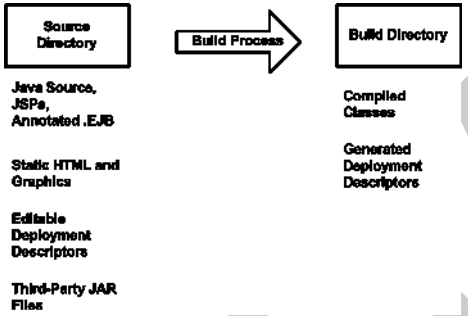
  Note: The split development directory structure does not provide support for developing new Resource Adapter components.

- 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,

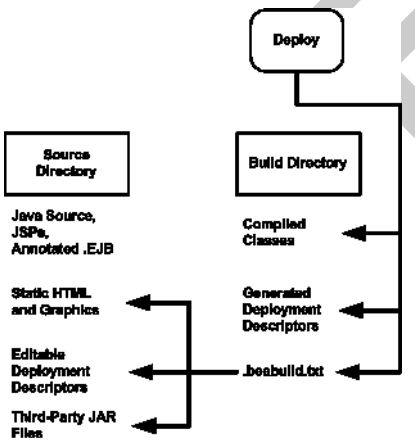
WebLogic Server then looks in the application's build directory for that resource. For example, if a deployment descriptor is generated during the build process, rather than stored with source code as an editable file, WebLogic Server obtains the generated file from the build directory.

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, "Split Directory Deployment," on page 3-4 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:

- `wlddcreate`—Generates basic deployment descriptor files for J2EE components in the source directory. See "Generating Deployment Descriptors Using wlddcreate" on page 4-2.
- `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 4-2.
- `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 4-5.
- `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 C-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 3-16.

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

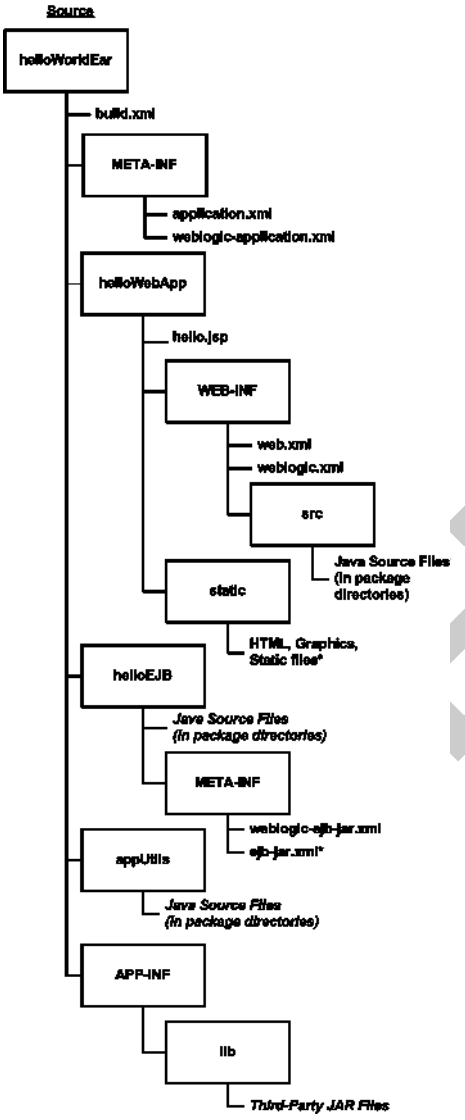
- "Enterprise Application Configuration" on page 3-9
- "Web Applications" on page 3-9
- "EJBs" on page 3-11
- "Shared Utility Classes" on page 3-12
- "Third-Party Libraries" on page 3-13

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.

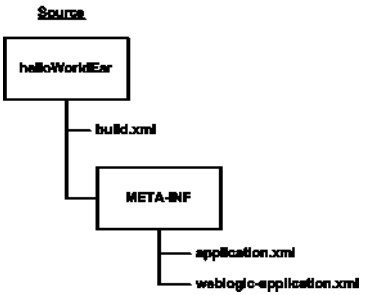
Figure 3-3 Overview of Enterprise Application Source Directory



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.

Notes: You can automatically generate Enterprise Application descriptors using the DDInit Java utility or `wlddcreate` Ant task. After adding J2EE module subdirectories to the EAR directory, execute the command:

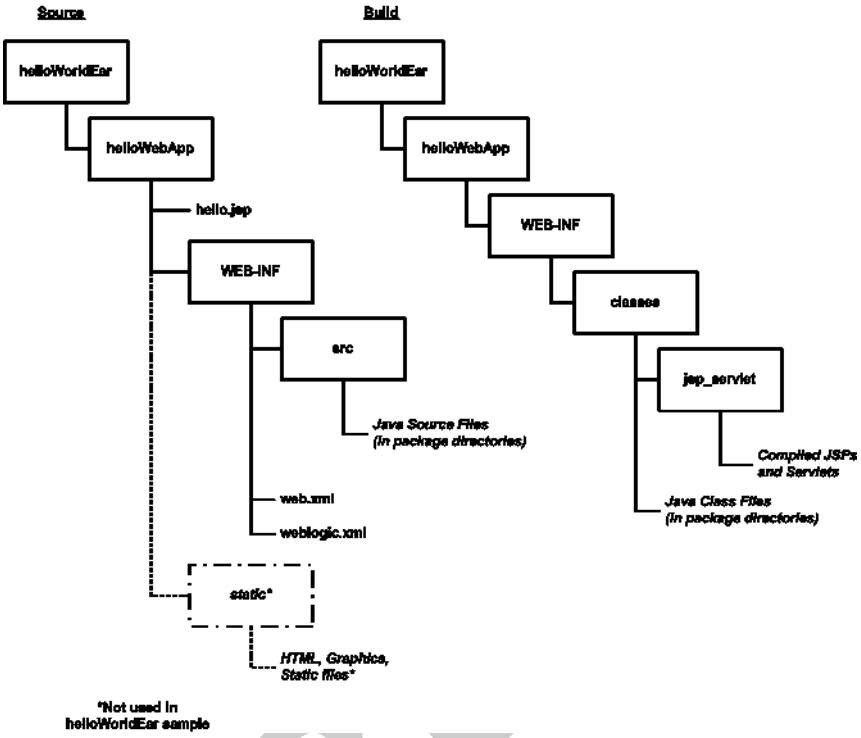
```
java weblogic.marathon.ddinit.EARInit \myEAR
```

For more information on DDInit, see "Using the WebLogic Server Java Utilities."

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:

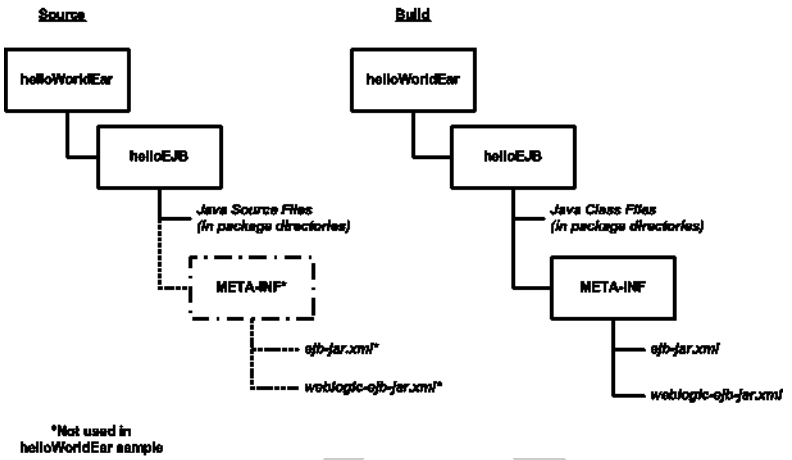
- `helloWebApp\` —The top level of the Web Application module can contain JSP files and static content such as HTML files and graphics used in the application. You can also store static files in any named subdirectory of the Web Application (for example, `helloWebApp\graphics` or `helloWebApp\static`.)
- `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.

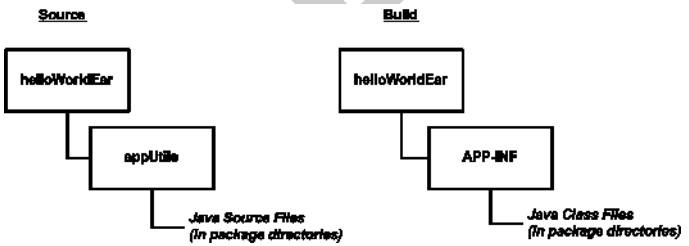
Organizing Shared Classes in a Split Development Directory

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



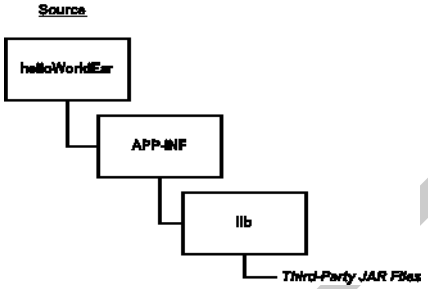
Place the source for Java utility classes in a named subdirectory of the top-level Enterprise Application directory. Beneath the named subdirectory, use standard package subdirectory conventions.

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 9-1.
- `-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 the
                      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.

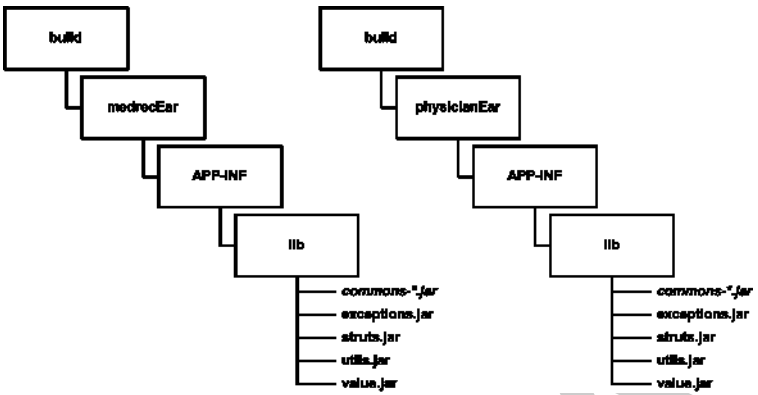
Organizing Libraries and Classes Shared by Multiple EARs

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:

Generating Deployment Descriptors Using wlddcreate

Note: The `wlddcreate` Ant task and `ddinit` commands are not provided in this Beta release of WebLogic Server.

The `wlddcreate` ant task provides a method for generating deployment descriptors for applications and application modules. It is an ant target provided as part of the generated `build.xml` file. It is an alternative to the `weblogic.marathon.ddinit` commands. The following is the `wlddcreate` target output:

```
<target name="descriptors" depends="compile" description="Generates
application and module descriptors">

<ddcreate dir="${dest.dir}" />

</target>
```

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 9-1.

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 9-1. The `library` element can be used with both `wlcompile` and `wlappc`, described in "Building Modules and Applications Using wlappc" on page 4-5.

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 Libraries in an Enterprise Application" on page 9-10. 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.

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 4-4.

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 9-1.
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 Libraries in an Enterprise Application" on page 9-10.
-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 C-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 6-3section 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.

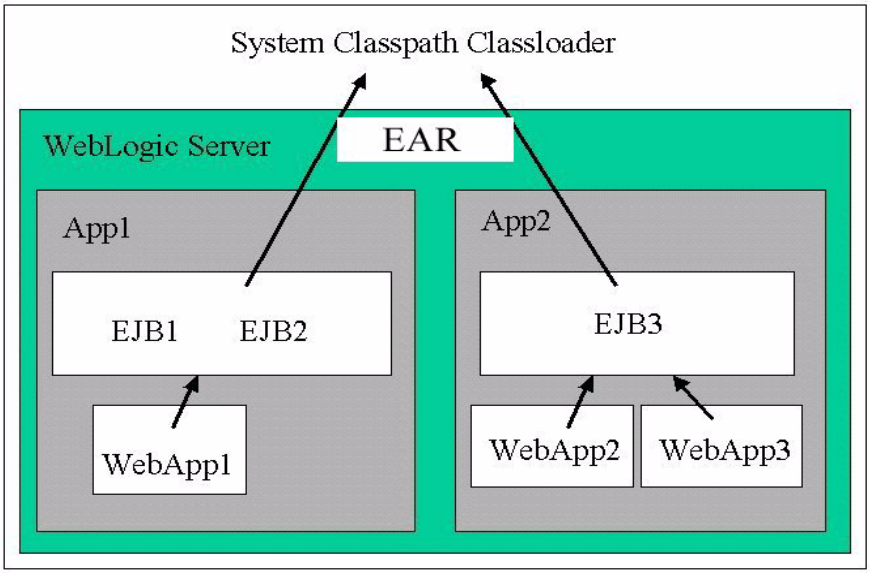
Application Classloader Hierarchy

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.

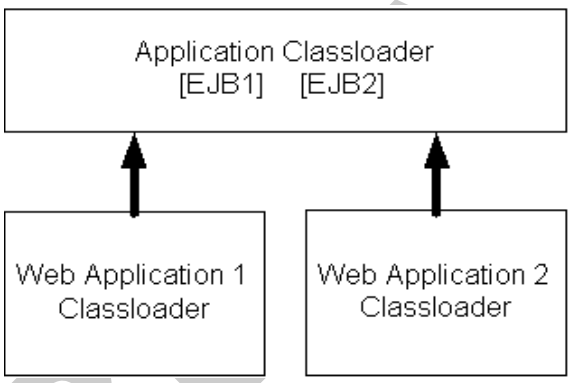
Custom Module Classloader Hierarchies

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`):

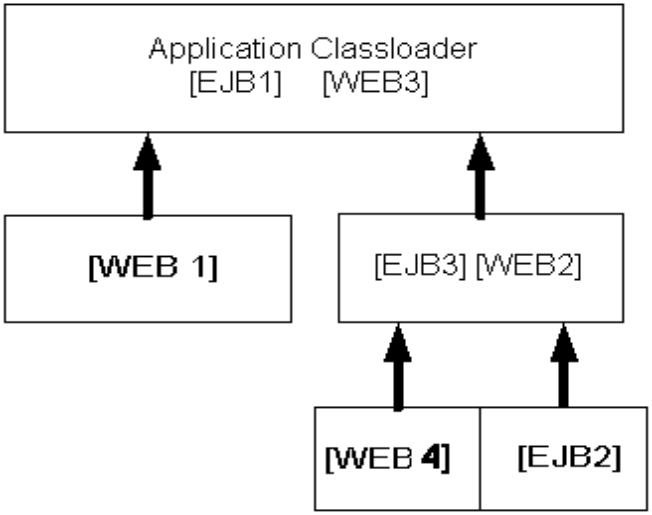
Listing 6-3 Example Classloader Declaration

```
<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

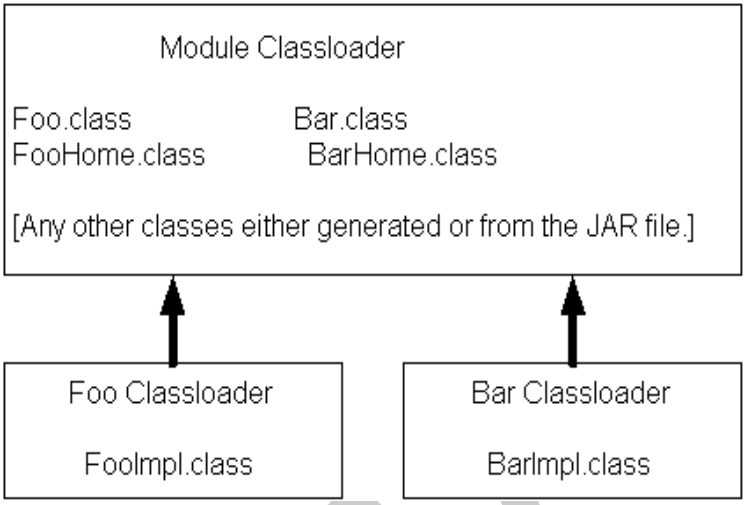
The development-use-only feature is intended for development use. Because updates are not atomic, this feature is not suitable for production use.

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`.

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 9-1.

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 prepends JAR files in 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 in Deploying Applications to WebLogic Server 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, "Valid and Invalid Characters," on page 7-6.

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 “.92Beta”:

```
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, "Read-Only Version Attributes in ApplicationMBean," on page 7-7 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

Exporting an Application for Deployment to New Environments

The following sections describe how to export an application's WebLogic Server deployment configuration to a custom deployment plan, which helps administrators easily deploy the application into non-development environments:

Overview of the Export Process

Exporting an application's deployment configuration is the process of creating a custom deployment plan that Administrators can use for deploying the application into new WebLogic Server environments. You distribute both the application deployment files and the custom deployment plan to deployers (for example, testing, staging, or production Administrators) who use the deployment plan as a blueprint for configuring the application for their environment.

An Administrator can install both the application and the custom deployment plan using the Administration Console, which validates the deployment plan and indicates when specific configuration properties need to be filled in before deployment.

See the Deployment Plan Reference and Schema in *Deploying Applications to WebLogic Server* for more information about deployment plans.

Goals for Exporting a Deployment Configuration

The primary goals in exporting a deployment configuration are:

1. **To expose the external resources requirements of the application as null variables in a deployment plan.** Any external resources required by the application are subject to change when the application is deployed to a different environment. For example, the JNDI names of datasources used in your development environment may be different from those used in testing or production. Exposing those JNDI names as variables makes it easy for deployers to use available resources or create required resources when deploying the application. Using empty (null) variables forces the deployer to fill in a valid resource name before the application can be deployed.
2. **To expose additional configurable properties, such as tuning parameters, as variables in a deployment plan.** Certain tuning parameters that are acceptable in a development environment may be unacceptable in a production environment. For example, it may suffice to accept default or minimal values for EJB caching on a development machine, whereas a production cluster would need higher levels of caching to maintain acceptable performance. Exporting selected tunables as deployment plan variables helps an Administrator focus on important tuning parameters when deploying the application. The Administration Console highlights tuning parameters exposed as variables in a deployment plan, but does not require a deployer to modify them before deployment.

Tools for Exporting a Deployment Configuration

BEA WebLogic Server provides the following tools to help you export an application's deployment configuration:

- `weblogic.Configure` creates a template deployment plan with null variables for selected categories of WebLogic Server deployment descriptors. This tool is recommended if you are beginning the export process and you want to create a template deployment plan with null variables for an entire class of deployment descriptors (see "Understanding Deployment Property Classifications" on page 8-3). You typically need to manually modify the deployment plan created by `weblogic.Configure`, either manually or using the Administration Console, to delete extraneous variable definitions or add variables for individual properties.
- The Administration Console updates or creates new deployment plans as necessary when you change configuration properties for an installed application. You can use the Administration Console to generate a new deployment plan or to add or override variables in an existing plan. The Administration Console provides greater flexibility than `weblogic.Configure`, because it allows you to interactively add or edit individual deployment descriptor properties in the plan, rather than export entire categories of descriptor properties.

Understanding Deployment Property Classifications

Each WebLogic Server deployment descriptor property (for all J2EE module descriptors as well as JDBC and JMS application modules) is formally classified into one of the following four categories:

- *Non-configurable* properties cannot be changed by an Administrator during a deployment configuration session. Non-configurable properties are used to describe application behavior that is fundamental to the basic operation of the application. For example, the `ejb-name` property is categorized as non-configurable, because changing its value also requires changing the EJB application code.
- *Dependency* properties resolve resource dependencies defined in the J2EE deployment descriptors. For example, if the J2EE descriptor for an EJB defines a datasource name that is used within the EJB code, the WebLogic Server descriptor uses a dependency property to bind the datasource name to an actual datasource configured in the target WebLogic Server domain.
- *Declaration* properties declare a resource that other applications can use. For example, the JNDI name of an EJB declares the EJB name that other applications or modules would use to access the EJB.
- *Configurable* properties are the remaining properties not classified as dependency or declaration properties. Generally configurable properties enable or configure WebLogic Server-specific features and tuning parameters for the deployed application. For example,

the WebLogic Server descriptor for an EJB might define the number of EJBs that WebLogic Server caches in memory.

These categories are used during the configuration export process to select properties to expose as variables in the deployment plan. For example, you can generate a new deployment plan containing variable definitions for all properties tagged as "dependencies" in an application's WebLogic Server deployment descriptors. The variables can then be easily changed by an Administrator deploying the application to an environment having different resource names.

All changeable descriptor properties (dependency, declaration, and configurable properties) are further classified as either *dynamic* or *non-dynamic* properties. Dynamic properties can be changed in a deployed application without requiring you to redeploy for the changes to take effect. Non-dynamic properties can be changed but require redeployment for the changes to take effect. The Administration Console identifies non-dynamic properties as a reminder for when redeployment is necessary.

Steps for Exporting an Application's Deployment Configuration

Exporting an application's deployment configuration typically involves the following procedures:

1. Staging Application Files for Export
2. Generating a Template Deployment Plan using weblogic.Configure
3. Customizing the Deployment Plan Using the Administration Console
4. Customizing the Deployment Plan by Hand
5. Validating the Exported Deployment Configuration

The sections that follow describe each procedure in detail.

Staging Application Files for Export

BEA recommends placing application files into an application installation directory before exporting the deployment configuration. When using an installation directory, generated configuration files, such as the deployment plan, are automatically copied to the `\plan` subdirectory during export.

To create an application installation directory:

1. Create a top-level installation directory for your application:

```
mkdir c:\exportapps\myApplication
```

2. Create `\app and \plan` subdirectories:

   ```
   mkdir c:\exportapps\myApplication\app
   ```

   ```
   mkdir c:\exportapps\myApplication\plan
   ```

3. Copy the complete application to be exported into the `\app` subdirectory. The application can be either in archive or exploded archive form:

   ```
   cp -r c:\dev\myApplication c:\exportapps\myApplication\app
   ```

   The `\app` directory must include the full application distribution, and can include the WebLogic Server descriptor files that you use for deployment to your development environment.

If you choose not to use an installation directory when exporting an application, BEA recommends using the `-plan` option to `weblogic.Configure` to specify the location and filename of the generated plan. By default, `weblogic.Configure` stores generated files in the *`TEMP`*`/weblogic-install/`*`application_name/config`* directory, where *`TEMP`* is the temporary directory for your environment. For Windows platforms, this means generated configuration files are stored in `C:\Documents and Settings\`*`username`*`\Local Settings\Temp\weblogic\install\myApplication.ear\config`. Use the `-plan` option to place generated files in a known location.

Generating a Template Deployment Plan using weblogic.Configure

The `weblogic.Configure` tool provides a quick and easy way to generate a template deployment plan with null variables for an entire category of deployment descriptors. BEA recommends using `weblogic.Configure` to generate a new deployment plan with null variables for all of an application's dependencies. This ensures that all global resources required for an application can be easily configured by Administrators who must deploy the application in a new environment.

When using an application staged in an installation root directory, the basic syntax for using `weblogic.Configure` is:

```
java weblogic.Configure -root install_root
     [-type (ear | war | jar | car | rar | jms | jdbc)]
     [-export (all | dependencies | declarations | dynamics)]
```

The `-type` option specifies the type of application or module you are exporting. If you omit this option, the tool attempts to derive the application or module type from the files available in the `/app` subdirectory.

The `-export` option specifies the category of WebLogic Server deployment descriptors for which you want to create variables. (See "Understanding Deployment Property Classifications" on page 8-3 for a description of each category.) For the purposes of generating a template deployment plan, you should use only the `-export dependencies` option, as this limits variables to external resources required by the application.

Note: Exporting `dynamics` creates null variables for every possible dynamically-configurable deployment property, which can result in a large number of variable definitions that may not be required for your application. Exporting `declarations` is generally not required, because declaration properties are typically associated with the basic functioning of the application and should not be changed before deployment to a new environment.

For example:

```
java weblogic.Configure -root c:\exportapps\myApplication -export
dependencies
```

With the above command, `weblogic.Configure` inspects all J2EE deployment descriptors in the selected application, and creates a deployment plan with null variables for all relevant WebLogic Server deployment properties that configure external resources for the application. Using this template deployment plan, an Administrator using the Administration Console would be directed to assign valid resource names for each null variable before the application could be deployed.

Customizing the Deployment Plan Using the Administration Console

The template deployment plan generated in "Generating a Template Deployment Plan using weblogic.Configure" on page 8-5 contains only those deployment properties that resolve external dependencies for the application. You will generally customize the template plan to add one or more WebLogic Server tuning properties for the application. The Administration Console enables you to easily add deployment plan variables for individual deployment descriptor properties as needed. To customize a deployment plan using the Administration Console:

1. Install the Exported Application and Template Deployment Plan
2. Add Variables for Selected Tuning Properties

3. Retrieve the Customized Deployment Plan

Install the Exported Application and Template Deployment Plan

To modify a deployment configuration using the Administration Console, you must first install the application and existing deployment plan:

1. Start the Administration Server on your development machine.
2. Access the Administration Console with a browser via http://*hostname*:*port*/console. For example: http://localhost:7001/console.
3. Login to the console using an Administration username and password (weblogic/weblogic by default).
4. Click Lock & Make Changes.
5. Select Deployments from the left pane to display the Summary of Deployments page.
6. On the Summary of Deployments page, click Install to begin the Install Application Assistant.
7. Use the links under the Location heading to locate the application installation directory you created for the exported application. Select the application installation directory and click Next.
8. On the Deployment identity page, note the name of the new installation, and click Finish. The Administration Console returns to the Summary of Deployments page and indicates that a new deployment has been installed.

Add Variables for Selected Tuning Properties

After installing the exported application, follow these steps to add new tuning properties to the deployment plan:

1. On the Summary of Deployments page, select the name of the application or module that you just installed.
2. Select the Deployment Plan tab. This tab contains two additional plans named Dependencies and Configuration.

   For Enterprise Applications, the Dependencies tab list each module within the application and enables you to review or set defaults for the dependency properties you created in

"Generating a Template Deployment Plan using weblogic.Configure" on page 8-5. The Configuration tab enables you to add tuning property values to the deployment plan.

3. Select the Deployment Plan > Configuration tab.
4. Edit tuning properties as necessary for the selected application or module.
5. Click Save and save the deployment plan to a known location.

Retrieve the Customized Deployment Plan

When you modify an application's deployment configuration using the Administration Console, your changes to deployment properties are stored in a WebLogic Server deployment plan and/or in generated WebLogic Server deployment descriptor files. If you modified only those deployment properties that were already defined as variables in the application's deployment plan, your changes are written back to a new version of the plan file. If you modified deployment properties that were not specified in the deployment plan, new variables are added to the plan.

When you configure an application that was installed from an installation directory, the Console stores generated configuration files in the `plan` subdirectory by default.

Customizing the Deployment Plan by Hand

In some cases you may need to edit a custom deployment plan manually, using a text editor. This may be necessary for the following reasons:

- You want to remove an existing deployment plan variable.
- You want to assign a null value to a generated variable in the plan.

Note: You cannot use the Administration Console to remove variables definitions from the deployment plan or assign a null value for a deployment property.

See the Deployment Plan Schema in *Deploying Applications to WebLogic Server* before manually editing deployment plan entries.

Removing Variables from a Deployment Plan

The `variable-definition` stanza in a deployment plan defines the names and values of variables used for overriding WebLogic Server deployment descriptor properties. The `module-override` stanza may contain one or more `variable-assignment` elements that define where a variable is applied to a given deployment descriptor. To remove a variable from a deployment plan, use a text editor to delete:

- the `variable` definition from the `variable-definition` stanza
- all `variable-assignment` elements that reference the deleted variable.

Assigning Null Variables to Require Administrator Input

Note: In this Beta release of WebLogic Server, null variables in a deployment plan are ignored. The Administration Console does not present null variables for additional configuration before deployment.

To assign a null value to an existing variable definition, simply remove any text value that is present in the `variable-definition` stanza. For example, change:

```
...
  <variable-definition>
    <variable>
      <name>SessionDescriptor_InvalidationIntervalSecs_11029744771850</name>
      <value>80</value>
    </variable>
  </variable-definition>
...
```

to:

```
...
  <variable-definition>
    <variable>
      <name>SessionDescriptor_InvalidationIntervalSecs_11029744771850</name>
      <value></value>
    </variable>
  </variable-definition>
...
```

Validating the Exported Deployment Configuration

Notes: The Administration Console in this Beta release of WebLogic Server does not support creating or modifying deployment plans for EJB modules, and does not support configuring external resources in a deployment plan. The Administration Console in this release can generate deployment plans that add or override tuning properties for an application.

In this Beta release, deployment plans can be used only with application deployment descriptors that use the new WebLogic Server schemas. Older, DTD-based deployment descriptors are not compatible with deployment plans.

The Administration Console automatically validates the deployment configuration for a newly-installed application or module. To validate a custom deployment plan that you have created during the export process:

1. Follow the steps under "Install the Exported Application and Template Deployment Plan" on page 8-7 to install the application or module with the final version of the custom deployment plan. The Administration Console automatically uses a deployment plan named `plan.xml` in the `plan` subdirectory of an installation directory, if one is available.
2. On the Summary of Deployments page, select the name of the application or module that you installed.
3. Select the Deployment Plan > Dependencies tab.
4. Verify that the dependencies configured for the deployed module are valid for the selected target servers.

Best Practices for Exporting a Deployment Configuration

Keep in mind these best practices when exporting an application's deployment configuration:

- The primary goal for exporting an application is to create null variables for all of an application's external resource dependencies. This ensures that deployers have the ability to assign resource names based on resources available in their target environment.
- Use `weblogic.Configure` only for exporting resource dependencies. Using `weblogic.Configure` to export other categories of deployment descriptor properties generally results in too many variables in the deployment plan.
- Use the Administration Console to add individual tuning property values to the deployment plan, or to validate a custom deployment plan.
- Neither the Administration Console nor `weblogic.Configure` allow you to remove variables from a plan or set null values for variables. Use a text editor when necessary to complete these tasks.

CHAPTER 9

Creating Shared J2EE Libraries and Optional Packages

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

Overview of J2EE Libraries and Optional Packages

Prior to WebLogic Server 9.0, multiple Enterprise Applications could not easily share a single J2EE module or 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 J2EE library feature in WebLogic Server 9.0 provides an easy way to share one or more J2EE modules among multiple Enterprise Applications. A J2EE library is a standalone J2EE module, multiple J2EE modules packaged in an Enterprise Application (EAR), or a single JAR file that is registered with the J2EE application container upon deployment. After the library has been registered, you can deploy Enterprise Applications that reference the library. Each referencing application receives a reference to the required library module(s) on deployment, and can use those modules as if they were packaged as part of the referencing application itself. The shared 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 J2EE library modules.

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 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 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 library or 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 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 Libraries and Dependent Applications" on page 9-15.

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 Libraries in an Enterprise Application" on page 9-10.

J2EE Libraries and Optional Packages Compared

Optional packages and 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 J2EE libraries and optional packages can specify required versions for the shared files.

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

- Optional packages are plain JAR files, whereas J2EE libraries can be plain JAR files, J2EE Applications, or standalone J2EE modules. This means that J2EE 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 can reference a J2EE library (using `weblogic-application.xml`).
- Optional packages can reference other optional packages, but J2EE libraries cannot reference other J2EE libraries.

In general, use J2EE libraries when you need to share one or more J2EE 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 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 *library* to refer to both J2EE libraries and optional packages. Optional packages are called out only when necessary.

Additional Information

For information about deploying and managing 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 J2EE library or optional package that you can share with multiple applications:

1. Assemble the 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.
2. Assemble optional package classes into a working directory.
3. Create and edit the `MANIFEST.MF` file for the library or package to specify the name and version string information.
4. Package the library or optional package for distribution and deployment.

The sections that follow describe each step in detail.

Assembling Shared J2EE Library Files

Almost any standalone J2EE module or Enterprise Application can be deployed as a shareable J2EE library. J2EE libraries have only the following restrictions:

- You cannot reference a shared J2EE library from another J2EE library.
- You must ensure that context roots in J2EE library Web Application do not conflict with context roots in the referencing Enterprise Application. If necessary, you can configure referencing applications to override a shared library's context root. See "Referencing Libraries in an Enterprise Application" on page 9-10.
- Shared libraries cannot be nested. For example, if you are deploying an EAR as a shared 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 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 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 module.

If you choose to deploy a 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 Libraries

The name and version information for a J2EE library or optional package are specified in the `META-INF/MANIFEST.MF` file. Table 9-1describes the valid J2EE library manifest attributes.

Table 9-1 Manifest Attributes for J2EE Libraries

Attribute	Description
`Extension-Name`	An optional string value that identifies the library or package name. Referencing applications must use the exact `Extension-Name` value to use the library or package. 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 this library or package. Referencing applications can optionally specify a required `Specification-Version` for a shared module; if the exact specification version is not available, deployment of the referencing application fails. The `Specification-Version` can use one of two different 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 module, a minimum version, or the latest available version. If you use the text format, referencing applications must specify the exact version of the library or package. The specification version for a library or module can also be set at the command-line when deploying the library, with some restrictions. See "Deploying Libraries and Dependent Applications" on page 9-15.
`Implementation-Version`	An optional String value that defines the code implementation version of this library or package. You can provide an `Implementation-Version` only if you have also defined a `Specification-Version`. `Implementation-Version` uses the same version formats as described in `Specification-Version` above. The implementation version for a library or module can also be set at the command-line when deploying the library, with some restrictions. See "Deploying Libraries and Dependent Applications" on page 9-15.

To specify attributes in a manifest file:

1. Open (or create) the manifest file using a text editor. For the example 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 library or package. For example:

   ```
   Extension-Name: myExtension
   ```

   Applications that reference the library or package 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 library or package. 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 or package from another application (see Table 9-1 on page 7)

   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 or package alongside older versions. See "Deploying Libraries and Dependent Applications" on page 9-15.

Packaging J2EE Libraries for Distribution and Deployment

If you are delivering the 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 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 J2EE library is packaged as a standard J2EE application or standalone module, you may also choose to export a J2EE library's deployment configuration to a deployment plan, as described in "Exporting an Application for Deployment to New Environments" on page 8-1. 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 Libraries in an Enterprise Application

A J2EE Application can reference a registered J2EE library using entries in the application's `weblogic-application.xml` deployment descriptor. Table 9-2 describes the XML elements that define a library reference.

Table 9-2 weblogic-application.xml Elements for Referencing a J2EE Library

Element	Description
`library-ref`	`library-ref` is the parent element in which you define a reference to a J2EE library. Enclose all other elements within `library-ref`.
`library-name`	A required string value that specifies the name of the J2EE library to use. `library-name` must exactly match the value of the `Extension-Name` attribute in the library's manifest file. (See Table 9-1 on page 7.)
`specification-version`	An optional String value that defines the required specification version of the 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 J2EE library having the exact same string value in the `Specification-Version` attribute in the library's manifest file. (See Table 9-1 on page 7.)
`implementation-version`	An optional String value that specifies the required implementation version of the 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 J2EE library having the exact same string value in the `Implementation-Version` attribute in the library's manifest file. (See Table 9-1 on page 7.)

Element	Description
`exact-match`	An optional boolean value that determines whether the application should use a J2EE library with a higher specification or implementation version than the configured value, if one is available. By default this element is false, meaning that WebLogic Server will use higher-versioned J2EE 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 J2EE library Web Application. Use this element if the context root of a J2EE library conflicts with the context root of a Web Application in the referencing J2EE application.

For example, this simple entry in the `weblogic-application.xml` descriptor references a 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` are 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 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 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</specification-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</specification-Version>
   <exact-match>true</exact-match>
</library-ref>
```

URIs for Standalone J2EE Module Libraries

When referencing the URI of a standalone J2EE library module, note that the module URI corresponds to the deployment name of the 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 Library modules 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 9-3 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 9-1 on page 7.) 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 9-1 on page 7.) 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
```


Integrating Libraries with the Split Development Directory Environment

The `BuildXMLGen` includes a `-librarydir` option to generate build targets that include one or more 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 library directories to include in the classpath for application builds. See "Building Applications in a Split Development Directory" on page 4-1.

Deploying Libraries and Dependent Applications

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 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.

After registering libraries, you can deploy applications and archives that depend on the shared files. 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.

Accessing Registered Library Information with LibraryRuntimeMBean

Each deployed 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 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.

Best Practices for Using J2EE Libraries

Keep in mind these best practices when developing J2EE Libraries and Optional Packages:

- Use J2EE Libraries when you want to share one or more J2EE modules with multiple Enterprise Applications.
- If you need to deploy a standalone J2EE module, such as an EJB JAR file, as a shared 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 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 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 libraries enables you to deploy multiple versions of the shared files for testing.
- 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
- When developing a Web Application for deployment as a 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 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 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 10

Programming Application Lifecycle Events

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

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.
- Startup classes only get preStart events.

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 10-1 illustrates how you override the `ApplicationLifecycleListener`. In this example, the public class `MyListener` extends `ApplicationLifecycleListener`.

Listing 10-1 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 10-2 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 10-2 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 10-3 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 10-3 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 10-4 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 10-5 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 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 1.1.3 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 WebLogic Server creates a single Session object and makes it available via JNDI to any module that needs it.

1. In the Administration Console, click on the Mail node in the left pane of the Administration Console.
2. Click Create a New Mail Session.
3. Complete the form in the right pane, as follows:
   - In the Name field, enter a name for the new session.
   - In the JNDIName field, enter a JNDI lookup name. Your code uses this string to look up the `javax.mail.Session` object.
   - In the Properties field, enter properties to configure the Session. The property names are specified in the JavaMail API Design Specification. JavaMail provides default values for each property, and you can override the values in the application code. The following table lists the properties you can set in this field.

Property	Description	Default
`mail.store.protocol`	Protocol for retrieving email. Example: `mail.store.protocol=imap`	The bundled JavaMail library is IMAP.
`mail.transport.protocol`	Protocol for sending email. Example: `mail.transport.protocol=smtp`	The bundled JavaMail library has supports for SMTP.
`mail.host`	The name of the mail host machine. Example: `mail.host=mailserver`	Local machine.
`mail.user`	Name of the default user for retrieving email. Example: `mail.user=postmaster`	Value of the `user.name` Java system property.

Property	Description	Default

`mail.`*`protocol`*`.host`	Mail host for a specific protocol. For example, you can set mail.SMTP.host and mail.IMAP.host to different machine names.
	Examples:
	`mail.smtp.host=mail.mydom.com`
	`mail.imap.host=localhost`
		Value of the `mail.host` property.

`mail.`*`protocol`*`.user`	Protocol-specific default user name for logging into a mailer server.
	Examples:
	`mail.smtp.user=weblogic`
	`mail.imap.user=appuser`
		Value of the `mail.user` property.

`mail.from`	The default return address. Examples: `mail.from=master@mydom.com`	`username@host`
`mail.debug`	Set to True to enable JavaMail debug output.	False

You can override any properties set in the Mail Session in your code by creating a `Properties` object containing the properties you want to override. See "Sending Messages with JavaMail" on page 11-4. 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 `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.

application.xml Deployment Descriptor Elements

For more information about `application.xml` deployment descriptor elements, please consult the J2EE 1.4 schema available at http://java.sun.com/xml/ns/j2ee/application_1_4.xsd.

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 application-scoped JDBC and JMS resources 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 Optional	Description
`<ejb>`	Optional	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-7.
`<xml>`	Optional	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-10.
<jdbc-connection-pool>		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-12.
<security>	Optional	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-24.

Element	Required Optional	Description
<application-param>		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. • `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. 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-25.

Element	Required Optional	Description

<classloader-structure>	Optional	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-25.

<listener>		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-25.
<startup>		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-26.
<shutdown>		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-26.

Element	Required Optional	Description
<module>		More information to come.
<library-ref>	Optional	A reference to a shared J2EE library.
<fair-share-request>	Optional	The name of a fair share request class.
<response-time-request>	Optional	The name of a response time request class.
<context-request>	Optional	The name of a context request class.
<max-threads-constraint>	Optional	The name of a maximum threads constraint.
<min-threads-constraint>	Optional	The name of a minimum threads constraint.
<capacity>	Optional	The name of a thread capacity definition.
<work-manager>	Optional	A work manager that can define request classes and thread constraints.
<application-admin-mode-trigger>	Optional	More information to come.

ejb

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

Element	Required Optional	Description

`<entity-cache>`		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-8.

`<start-mbds-with-application`	Optional	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 Optional	Description
`<entity-cache-name>`		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	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 Optional	Description
<max-cache-size>	Optional	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. `bytes | megabytes`—The size of an entity cache in terms of memory size, expressed in bytes or megabytes. Used in the `max-cache-size` element.
<caching-strategy>	Optional	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>

xml

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

Element	Required Optional	Description
`<parser-factory>`	Optional	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-10.
`<entity-mapping>`	Optional	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-11.

parser-factory

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

Element	Required Optional	Description
`<saxparser-factory>`	Optional	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

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

jdbc-connection-pool

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

Element	Required Optional	Description
`<data-source-jndi-name>`		Specifies the JNDI name in the application-specific JNDI tree.
`<connection-factory>`		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-13.
<pool-params>	Optional	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-14.
<driver-params>	Optional	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-21.

connection-factory

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

Element	Required Optional	Description
`<factory-name>`	Optional	Specifies the name of a `JDBCDataSourceFactoryMBean` in the `config.xml` file.
`<connection-properties>`	Optional	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> <param-name>foo</param-name> <param-value>xyz</param-value> </parameter>

pool-params

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

Element	Required Optional	Description
<size-params>	Optional	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 Optional	Description
<xa-params>	Optional	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. • 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.

Element	Required Optional	Description
<xa-params> Continued...	Optional	• 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. • 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>

Element	Required Optional	Description
<login-delay-seconds>	Optional	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	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 Optional	Description
<connection-check-params>	Optional	• 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.
<connection-check-params> Continued...	Optional	• `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.

Element	Required Optional	Description
<jdbcxa-debug-level>	Optional	This is an internal setting.
<remove-infected-connections-enabled>	Optional	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 Optional	Description

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

Element	Required Optional	Description
<prepared-statement	Optional	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.
<row-prefetch-enabled>	Optional	
<row-prefetch-size>	Optional	
<stream-chunk-size>	Optional	

security

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

Element	Required Optional	Description
`<realm-name>`	Optional	Names a security realm to be used by the application. If none is specified, the system default realm is used
`<security-role-assignment>`		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 Optional	Description
`<description>`	Optional	Provides a description of the application parameter.
<param-name>		Defines the name of the application parameter.
<param-value>		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 Optional	Description
`<module-ref>`		Zero or more. The following table describes the elements you can define within a `module-ref` element. `module-uri`—Zero or more. Defined within the `module-ref` element.

listener

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

Element	Required Optional	Description
`<listener-class>`		Name of the user's implementation of `ApplicationLifecycleListener`.
<listener-uri>	Optional	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.

Element	Required Optional	Description
`<startup-class>`		Defines the name of the class to be run when the application is being deployed.
<startup-uri>	Optional	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.

Element	Required Optional	Description
`<shutdown-class>`		Defines the name of the class to be run when the application is undeployed.
<shutdown-uri>	Optional	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.

weblogic-application.xml Schema

```
<?xml version="1.0" encoding="UTF-8"?>

<schema targetNamespace="http://www.bea.com/ns/weblogic/90"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:wls="http://www.bea.com/ns/weblogic/90" elementFormDefault="qualified"
attributeFormDefault="unqualified">

<include schemaLocation="weblogic-j2ee.xsd"/>

<element name="weblogic-application" type="wls:weblogic-applicationType"/>

<complexType name="weblogic-applicationType">

<sequence>
```

```
<element name="ejb" type="wls:ejbType" minOccurs="0"/>

<element name="xml" type="wls:xmlType" minOccurs="0"/>

<element name="jdbc-connection-pool" type="wls:jdbc-connection-poolType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="security" type="wls:securityType" minOccurs="0"/>

<element name="application-param" type="wls:application-paramType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="classloader-structure" type="wls:classloader-structureType"
minOccurs="0"/>

<element name="listener" type="wls:listenerType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="startup" type="wls:startupType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="shutdown" type="wls:shutdownType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="module" type="wls:weblogic-moduleType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="library-ref" type="wls:library-refType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="fair-share-request" type="wls:fair-share-request-classType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="response-time-request"
type="wls:response-time-request-classType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="context-request" type="wls:context-request-classType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="max-threads-constraint" type="wls:max-threads-constraintType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="min-threads-constraint" type="wls:min-threads-constraintType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="capacity" type="wls:capacityType" minOccurs="0"
maxOccurs="unbounded"/>

<element name="work-manager" type="wls:work-managerType" minOccurs="0"
maxOccurs="unbounded"/>
```

```
      <!-- Configure the number of stuck threads needed to bring the app into
admin mode -->

      <element name="application-admin-mode-trigger"
type="wls:application-admin-mode-triggerType" minOccurs="0" maxOccurs="1"/>

<!-- The ejb element 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.

          Used in: weblogic-application

        -->

<!-- The xml element contains information about parsers, and entity mappings

          for xml processing that is specific to this application.

        -->

<!-- The jdbc-connection-pool element is the root element for an

          application scoped JDBC pool declaration.

        -->

<!-- The jms element specifies the URI of a JMS destination group module-->

</sequence>

</complexType>

<complexType name="ejbType">

<sequence>

<element name="entity-cache" type="wls:application-entity-cacheType"
minOccurs="0" maxOccurs="unbounded"/>

<element name="start-mdbs-with-application" type="wls:true-falseType"
minOccurs="0"/>

<!--

        The entity-cache element is used to define a named application level

        cache that will be used to cache entity EJB instances at runtime.

        Individual entity beans refer to the application level cache that they
```

```
        want to use using the cache's 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
        doesn't specify its own cache in the weblogic-ejb-jar.xml descriptor.
        Two default caches with names, 'ExclusiveCache' and 'MultiVersionCache'
        are used for this purpose.  An application may explicitly define these
        default caches if it wants to specify non-default values for
        their settings.  Note, however, 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.

        Used in: ejb
        Example:
        <entity-cache>
          <entity-cache-name>ExclusiveCache</entity-cache-name>
          <max-cache-size>
          <megabytes>50</megabytes>
          </max-cache-size>
        </entity-cache>

      -->
<!--
        This allows user to configure ejb container to start mdbs with
```

```
        application.  If set to "true", container starts mdbs as part of
        application.  If set to "false" container keeps mdbs in a queue and
        server will start them as soon as it started listening on ports.

        Example:
        <start-mdbs-with-application>true</start-mdbs-with-application>

      -->
</sequence>
</complexType>
<complexType name="application-entity-cacheType">
<sequence>
<element name="entity-cache-name" type="string"/>
<choice minOccurs="0">
<element name="max-beans-in-cache" type="int"/>
<element name="max-cache-size" type="wls:max-cache-sizeType"/>
<!--
                Maximum number of entity beans that are allowed in the cache.
If the
              limit is reached, beans may be passivated.  If 0 is specified, then
               there is no limit.  This mechanism does not take into account the
                  actual ammount of memory that different entity beans require.

                  Used in: entity-cache

                  Default value: 1000
                  -->
<!--
```

```
            The max-cache-size element is 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.

            Used in: entity-cache
         -->
</choice>
<element name="caching-strategy" type="string" minOccurs="0"/>
<!--
      The entity-cache-name element 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.

      Used in: entity-cache

      Example:
        <entity-cache-name>ExclusiveCache</entity-cache-name>
    -->
<!--
          The caching-strategy element 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 have one of the following values:

          - Exclusive: An 'Exclusive' cache 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: A 'MultiVersion' cache 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>
        -->
</sequence>
</complexType>
<complexType name="max-cache-sizeType">
<choice>
<element name="bytes" type="int"/>
<element name="megabytes" type="int"/>
<!--
        The bytes element is used to specify the maximum size of an
```

```
         application level entity cache in bytes.

         Example:
         <bytes>15000000</bytes>
        -->
<!--
          The megabytes element is used to specify the maximum size of an entity
          bean cache in megabytes.
        -->
</choice>
</complexType>
<!--
       xml element contains information about parsers, and entity mappings
       for xml processing that is specific to this application.
     -->
<complexType name="xmlType">
<sequence>
<element name="parser-factory" type="wls:parser-factoryType" minOccurs="0"/>
<element name="entity-mapping" type="wls:entity-mappingType" minOccurs="0"
maxOccurs="unbounded"/>
<!--
          The parser-factory element is used to configure factory classes
          for both SAX and DOM style parsing and for XSLT transformations
          for the enterprise application

          Example
            <parser-factory>
```

```
<saxparser-factory>weblogic.xml.babel.jaxp.SAXParserFactoryImpl</saxparser-fac
tory>

<document-builder-factory>org.apache.xerces.jaxp.DocumentBuilderFactoryImpl</d
ocument-builder-factory>

<transformer-factory>org.apache.xalan.processor.TransformerFactoryImpl</transf
ormer-factory>
          </parser-factory>


       -->
<!--
          This element is used to specify entity mapping.  This mapping
          determines alternative entityuri for given public / system id.
          Default place to look for these entityuri is "lib/xml/registry"
          directory.
        -->
</sequence>
</complexType>
<complexType name="parser-factoryType">
<sequence>
<element name="saxparser-factory" type="string" minOccurs="0"/>
<element name="document-builder-factory" type="string" minOccurs="0"/>
<element name="transformer-factory" type="string" minOccurs="0"/>
</sequence>
</complexType>
<complexType name="entity-mappingType">
<sequence>
<element name="entity-mapping-name" type="string"/>
```

```
<element name="public-id" type="string" minOccurs="0"/>
<element name="system-id" type="string" minOccurs="0"/>
<element name="entity-uri" type="string" minOccurs="0"/>
<element name="when-to-cache" type="string" minOccurs="0"/>
<element name="cache-timeout-interval" type="int" minOccurs="0"/>
<!--
        This element specifies the name for this entity mapping.
        -->
<!--
        This element specifies the public id of the mapped entity.
        -->
<!--
        This element specifies the system id of the mapped entity.
        -->
<!--
        This element specifies the entityuri for the mapped entity.
        -->
<!--
        This element specifies the  when-to-cache for the mapped entity.
        -->
<!--
        This element specifies cache time out interval  for the mapped entity.
        -->
</sequence>
</complexType>
<!--
          The jdbc-connection-pool element is the root element for an
```

```
          application scoped JDBC pool declaration.
        -->
<complexType name="jdbc-connection-poolType">
<sequence>
<element name="data-source-jndi-name" type="string"/>
<element name="connection-factory" type="wls:connection-factoryType"/>
<element name="pool-params" type="wls:application-pool-paramsType"
minOccurs="0"/>
<element name="driver-params" type="wls:driver-paramsType" minOccurs="0"/>
<element name="acl-name" type="string" minOccurs="0"/>
<!-- A JNDI name in the application specific JNDI tree -->
<!--
        Defines initialization parameters for a connection factory which is
        used to construct connection pools.
        -->
<!--
          DEPRECATED: The acl-name is deprecated
         -->
</sequence>
</complexType>
<complexType name="connection-factoryType">
<sequence>
<element name="factory-name" type="string" minOccurs="0"/>
<element name="connection-properties" type="wls:connection-propertiesType"
minOccurs="0"/>
<!--
        The name of a JDBCDataSourceFactoryMBean in config.xml
        -->
```

```
<!--
        The connection params define overrides for default connection factory
        settings
        -->
</sequence>
</complexType>
<complexType name="connection-propertiesType">
<sequence>
<element name="user-name" type="string" minOccurs="0"/>
<element name="password" type="string" minOccurs="0"/>
<element name="url" type="string" minOccurs="0"/>
<element name="driver-class-name" type="string" minOccurs="0"/>
<element name="connection-params" type="wls:connection-paramsType"
minOccurs="0" maxOccurs="unbounded"/>
<!--
        user-name is an optional element which is used to override UserName in
        the JDBCDataSourceFactoryMBean.
        -->
<!--
        password is an optional element which is used to override Password in
        the JDBCDataSourceFactoryMBean.
        -->
<!--
        url is an optional element which is used to override URL in the
        JDBCDataSourceFactoryMBean.
        -->
<!--
        driver-class-name is an optional element which is used to override
```

```
        DriverName in the JDBCDataSourceFactoryMBean
        -->
<!--
          The connection-params element is used to set parameters which will be
           passed to the driver when making a connection.

           Example:
            <connection-params>
             <parameter>
              <param-name>foo</param-name>
              <param-value>xyz</param-value>
             </parameter>

           -->
</sequence>
</complexType>
<complexType name="connection-paramsType">
<sequence>
<element name="parameter" type="wls:parameterType" minOccurs="0"
maxOccurs="unbounded"/>
</sequence>
</complexType>
<complexType name="parameterType">
<sequence>
<element name="description" type="string" minOccurs="0"/>
<element name="param-name" type="string"/>
<element name="param-value" type="string"/>
<!--
```

```
          The param-name is the description of a parameter that is passed to one
          of the application components.

          Used in: connection-params/parameter and application-param
        -->
<!--
          The param-name is the name of a parameter that is passed to one
          of the application components.

          Used in: connection-params/parameter and application-param
        -->
<!--
          The param-name is the value of a parameter that is passed to one
          of the application components.

          Used in: connection-params/parameter and application-param
          -->
</sequence>
</complexType>
<!--
        The pool-params element defines parameters that affect the behavior of
        the pool
        -->
<complexType name="application-pool-paramsType">
<sequence>
<element name="size-params" type="wls:size-paramsType" minOccurs="0"/>
<element name="xa-params" type="wls:xa-paramsType" minOccurs="0"/>
```

```
<element name="login-delay-seconds" type="int" minOccurs="0"/>

<element name="leak-profiling-enabled" type="wls:true-falseType"
minOccurs="0"/>

<element name="connection-check-params" type="wls:connection-check-paramsType"
minOccurs="0"/>

<element name="jdbcxa-debug-level" type="int" minOccurs="0"/>

<element name="remove-infected-connections-enabled" type="wls:true-falseType"
minOccurs="0"/>

<!--

          Defines a login delay for connection requests. This is used to

          prevent rapid requests for JDBC connections from overwhelming

          the DBMS server.

        -->

<!--

          If set to true, the system will print exceptions to the log file

          when code using a connection pool fails to return the connection

          to the pool before dereferencing it.

        -->

<!--

          This is an internal setting

        -->

<!--

          If set to true, connection will be removed from pool if application

          asks for the underlying vendor connection object.

        -->

</sequence>

</complexType>

<!--

        The size-params element defines parameters that affect the number of
```

```
        connections in the pool
        -->
<complexType name="size-paramsType">
<sequence>
<element name="initial-capacity" type="int" minOccurs="0"/>
<element name="max-capacity" type="int" minOccurs="0"/>
<element name="capacity-increment" type="int" minOccurs="0"/>
<element name="shrinking-enabled" type="wls:true-falseType" minOccurs="0"/>
<element name="shrink-period-minutes" type="int" minOccurs="0"/>
<element name="shrink-frequency-seconds" type="int" minOccurs="0"/>
<element name="highest-num-waiters" type="int" minOccurs="0"/>
<element name="highest-num-unavailable" type="int" minOccurs="0"/>
<!--
          An integer value representing the initial size of the pool
        -->
<!--
          An integer value representing the maximum size of the pool
        -->
<!--
          An integer value representing the number of connections that
          will be added to a pool when a request is made to a pool
          that has no available connections.
        -->
<!--
          This can be set to true|false. If set to true, then
          the pool having more than the initial capacity of
          connections will shrink back towards the initial size
```

```
          when connections become unused.
        -->
<!--
          Number of seconds to wait before shrinking a
          connection pool that has incrementally increased to meet demand.
        -->
<!--
          defines the maximum number of threads that will wait for a connection
          before an exception is thrown to the requestor.
          -->
<!--
          defines the maximum number of connections being refreshed
        -->
</sequence>
</complexType>
<!-- params for XA data sources -->
<complexType name="xa-paramsType">
<sequence>
<element name="debug-level" type="int" minOccurs="0"/>
<element name="keep-conn-until-tx-complete-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="end-only-once-enabled" type="wls:true-falseType" minOccurs="0"/>
<element name="recover-only-once-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="tx-context-on-close-needed" type="wls:true-falseType"
minOccurs="0"/>
<element name="new-conn-for-commit-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="prepared-statement-cache-size" type="int" minOccurs="0"/>
```

```
<element name="keep-logical-conn-open-on-release" type="wls:true-falseType"
minOccurs="0"/>

<element name="local-transaction-supported" type="wls:true-falseType"
minOccurs="0"/>

<element name="resource-health-monitoring-enabled" type="wls:true-falseType"
minOccurs="0"/>

<element name="xa-set-transaction-timeout" type="wls:true-falseType"
minOccurs="0"/>

<element name="xa-transaction-timeout" type="int" minOccurs="0"/>

<element name="rollback-localtx-upon-connclose" type="wls:true-falseType"
minOccurs="0"/>

<!-- These are all translated from JDBCConnectionPoolMBean -->

<!--

          Sets the Debug Level for XA Drivers

          Default value: 0

          Example:

             <debug-level>3</debug-level>


          Since: WLS 7.0

        -->

<!--


          Determines whether the XA connection pool associates the same XA

          connection with the distributed transaction until the transaction

          completes.  This property applies to XA connection pools only, and

          is ignored for non-XA driver.  Its intention is to workaround

          specific problems with third party vendor's XA driver.


          Default value: false

          Used in: xa-params
```

```
          Example:
            <keep-conn-until-tx-complete-enabled>
             true
            </keep-conn-until-tx-complete-enabled>
          Since: WLS 7.0
        -->
<!--

          Determines whether XAResource.end() will be called only once for
          each pending XAResource.start().  e.g. the XA driver will not be
          called XAResource.end(TMSUSPEND), XAResource.end(TMSUCCESS)
          successively.

          This property applies to XA connection pools only, and is ignored
          for non-XA driver.  Its intention is to workaround specific
          problems with third party vendor's XA driver.

          Default value: false
          Used in: xa-params
          Example:
            <end-only-once-enabled>
              true
            </end-only-once-enabled>

          Since: WLS 7.0

        -->
```

```
<!--

        Declares whether JTA TM should call recover on the resource once
        only.

        This property applies to XA connection pools only, and is ignored
        for non-XA driver.  Its intention is to workaround specific problems
        with third party vendor's XA driver.

        Default value: false
        Used in: xa-params
        Example:
          <recover-only-once-enabled>
            true
          </recover-only-once-enabled>

        Since: WLS 7.0

      -->
<!--

        Defines whether the XA driver requires a distributed transaction
        context when closing various JDBC objects, e.g. result sets,
        statements, connections etc.  If it is specified to true, SQL
        exceptions that are thrown while closing the JDBC objects in no
        transaction context will be swallowed.  This property applies to XA
        connection pools only, and is ignored for non-XA driver.  Its
```

```
          intention is to workaround specific problems with third party
          vendor's XA driver.

          Default value: false
          Used in: xa-params
          Example:
            <tx-context-on-close-needed>
              true
            </tx-context-on-close-needed>
          Since: WLS 7.0

        -->
<!--

          Defines whether a dedicated XA connection is used for
          commit/rollback processing of a particular distributed transaction.

          This property applies to XA connection pools only, and is ignored
          for non-XA driver.  Its intention is to workaround specific
          problems with third party vendor's XA driver.


          Default value: false
          Used in: xa-params
          Example:
            <tx-context-on-close-needed>
              true
```

```
      </tx-context-on-close-needed>
    Since: WLS 7.0

  -->
<!--

    The maximum number of prepared statements cached by this particular
    XA connection pool.  If the value is 0, caching is turned off.

    This property applies to XA connection pools only, and is ignored
    for non-XA driver.

    Default value: 0
    Used in: xa-params
    Example:
      <prepared-statement-cache-size>
        3
      </prepared-statement-cache-size>

    Since: WLS 7.0

  -->
<!--
    Default value:
    Used in: xa-params
    Example:
      <keep-logical-conn-open-on-release>
```

```
              true
            </keep-logical-conn-open-on-release>
          Since: WLS 7.0

        -->
<!--
          Default value:
          Used in: xa-params
          Example:
             <local-transaction-supported>
               true
             </local-transaction-supported>
          Since: WLS 7.0

        -->
<!--
          Default value:
          Used in: xa-params
          Example:
             <resource-health-monitoring-enabled>
               true
             </resource-health-monitoring-enabled>
          Since: WLS 7.0

        -->
<!--
          Default value:
```

```
      Used in: xa-params
      Example:
         <xa-set-transaction-timeout>
           true
         </xa-set-transaction-timeout>
      Since: WLS 8.1

    -->
<!--
      When the xa-set-transaction-timeout value is set to true,
      the TM will invoke setTransactionTimeout on the resource before
      calling XAResource.start.  The TM will pass the global transaction
      timeout value.  If this attribute is set to a value greater than 0,
    then this value will be used in place of the global transaction timeout.

      Default value: 0
      Used in: xa-params
      Example:
         <xa-transaction-timeout>
           30
         </xa-transaction-timeout>
      Since: WLS 8.1

    -->
<!--
      When the rollback-localtx-upon-connclose element is true,
      WLS connection pool will call 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>
            Since: WLS 8.1

         -->
</sequence>
</complexType>
<!--
         The connection-check-params define whether, when and how connections
         in a pool will be checked to make sure they are still alive.
         -->
<complexType name="connection-check-paramsType">
<sequence>
<element name="table-name" type="string" minOccurs="0"/>
<element name="check-on-reserve-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="check-on-release-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="refresh-minutes" type="int" minOccurs="0"/>
<element name="check-on-create-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="connection-reserve-timeout-seconds" type="int" minOccurs="0"/>
```

```
<element name="connection-creation-retry-frequency-seconds" type="int"
minOccurs="0"/>
<element name="inactive-connection-timeout-seconds" type="int" minOccurs="0"/>
<element name="test-frequency-seconds" type="int" minOccurs="0"/>
<element name="init-sql" type="string" minOccurs="0"/>
<!-- table-name defines a table in the schema that can be queried -->
<!--
        If on-reserve is "true", then the connection will be tested each time
        before its handed out to a user.
        -->
<!--
        If on-release is "true", then the connection will be tested each time
        a user returns a connection to the pool.
        -->
<!--
        If refresh-minutes is defined then a trigger will be fired
        periodically (based on the number of minutes specified). This trigger
        will check each connection in the pool to make sure it is still valid.
        -->
<!--
        If on-create is "true", then the connection will be tested when it is
        created.
        -->
<!--
        Number of seconds after which the call to reserve a connection from the
        pool will timeout.
        -->
<!--
```

```
        The frequency of retry attempts by the pool to establish connections
        to the database.
        -->
<!--
        # seconds of inactivity after which reserved connections will forcibly
        be released back into the pool.
        -->
<!--
        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 defines a sql query that will run when a connection
        is created
        -->
</sequence>
</complexType>
<!--
        the driver params are used to set behavior on weblogic drivers
        -->
<complexType name="driver-paramsType">
<sequence>
<element name="statement" type="wls:statementType" minOccurs="0"/>
<element name="prepared-statement" type="wls:prepared-statementType"
minOccurs="0"/>
<element name="row-prefetch-enabled" type="wls:true-falseType" minOccurs="0"/>
```

```
<element name="row-prefetch-size" type="int" minOccurs="0"/>
<element name="stream-chunk-size" type="int" minOccurs="0"/>
<!-- These are all translated from JDBCConnectionPoolMBean -->
<!--
           Controls 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 will ignore
           this setting and not use row prefetching when the client and
           WebLogic Server are in the same JVM.
          -->
<!--
        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 will increase performance, until a particular value
           is reached.  At that point further increases do not result in any
           significant performance increase.  Very rarely will increased
           performance result from exceeding 100 rows.  The default value
           should be reasonable for most situations.

           The default for this setting is 48
           The range for this setting is 2 to 65536
          -->
<!--
          Data chunk size for steaming datatypes.  Streaming datatypes (for
          example resulting from a call to <code>getBinaryStream()</code>)
```

```
          will be pulled in StreamChunkSize sized chunks from WebLogic
          Server to the client as needed.
          -->
</sequence>
</complexType>
<!--
          Default value:
          Used in: driver-params
          Example:
             <statement>
               <profiling-enabled>true</profiling-enabled>
             </statement>
          Since: WLS 7.0

        -->
<complexType name="statementType">
<sequence>
<element name="profiling-enabled" type="wls:true-falseType" minOccurs="0"/>
</sequence>
</complexType>
<!--
          Default value:
          Used in: driver-params
          Example:
             <prepared-statement>
               <profiling-enabled>true</profiling-enabled>
               <cache-size>3</cache-size>
```

```
                <parameter-logging-enabled>true</parameter-logging-enabled>
                <max-parameter-length>10</max-parameter-length>
              </prepared-statement>
            Since: WLS 7.0


        -->
<complexType name="prepared-statementType">
<sequence>
<element name="profiling-enabled" type="wls:true-falseType" minOccurs="0"/>
<element name="cache-profiling-threshold" type="int" minOccurs="0"/>
<element name="cache-size" type="int" minOccurs="0"/>
<element name="parameter-logging-enabled" type="wls:true-falseType"
minOccurs="0"/>
<element name="max-parameter-length" type="int" minOccurs="0"/>
<element name="cache-type" type="int" minOccurs="0"/>
</sequence>
</complexType>
<!--
        This element specifies security information for the application
        -->
<complexType name="securityType">
<sequence>
<element name="realm-name" type="string" minOccurs="0"/>
<element name="security-role-assignment"
type="wls:application-security-role-assignmentType" minOccurs="0"
maxOccurs="unbounded"/>
<!--
        This element names a security realm that will be used by the
        application. If no specified, then the system default realm will be
```

```
        used
        -->
</sequence>
</complexType>
<!--
       The security-role-assigment declares a mapping between an application wide
        security role and one or more principals in the WebLogic server.

        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>

        Used in: security
        -->
<complexType name="application-security-role-assignmentType">
```

```
<sequence>
<element name="role-name" type="string"/>
<element name="principal-name" type="string" maxOccurs="unbounded"/>
<!--
        The role-name element contains the name of a security role.

        Used in: security-role-assignment
        -->
<!--
        The principal-name element contains the name of a principal.

        Used in: security-role-assignment
        -->
</sequence>
</complexType>
<!--
          The application-param element is used to specify untyped parameters
that affect
          the behavior of container instances related to the application.
Currently, the
          following parameters are supported:

          The following parameters in weblogic-application.xml can determine
          the default encoding to be used for both request and response.

          webapp.encoding.default - can be set to a string representing an
              encoding supported by the JDK. If set, this will define the
```

```
            default encoding used to process servlet requests and responses.
            This setting is ignored if webapp.encoding.usevmdefault is set
            to true. This value is also overridden for requests streams
            by the input-charset element of weblogic.xml and for response
            streams by the contentType header of the request.

        webapp.encoding.usevmdefault - can be set to true or false. If true,
            the system property file.encoding will be 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 which may be set to true or false. If set to true,
            then 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>


        -->
```

```
<complexType name="application-paramType">
<sequence>
<element name="description" type="string" minOccurs="0"/>
<element name="param-name" type="string"/>
<element name="param-value" type="string"/>
</sequence>
</complexType>
<!--
          A classloader-structure element allows you to define the organization
          of classloaders for this application. The declaration represents a
        tree structure which represents the classloader hierarchy and associates
          specific modules with particular nodes. A module's classes will be
          loaded by the classloader that its associated with in this structure.

          Example:
           <classloader-structure>
             <module-ref>
              <module-uri>ejb1.jar</module-uri>
              <module-uri>ejb2.jar</module-uri>
              <classloader-structure>
                <module-uri>ejb3.jar</module-uri>
              </classloader-structure>
           </classloader-structure>

        In this example, the classloader structure would look like the following
         diagram:
```

```
            +====================+
            |root classloader    |
            | ejb1 classes and   |
            | ejb2 classes       |
            |                    |
            +====================+
                      ^
                      |
            +====================+
            |child classloader   |
            | ejb3 classes       |
            |                    |
            +====================+


        This allows for arbitrary nestings, however, for this version the depth
        is restricted to three levels.


        -->
<complexType name="classloader-structureType">
<sequence>
<element name="module-ref" type="wls:module-refType" minOccurs="0"
maxOccurs="unbounded"/>
<element name="classloader-structure" type="wls:classloader-structureType"
minOccurs="0" maxOccurs="unbounded"/>
</sequence>
</complexType>
<complexType name="module-refType">
<sequence>
```

```
<element name="module-uri" type="string"/>

</sequence>

</complexType>

<!--

          The listener element is used to register user defined application
lifecycle

          listeners. These are classes that extend the abstract base class

          weblogic.application.ApplicationLifecycleListener. The listener-class

          element is the name of the users implementation of
ApplicationLifecycleListener.

          The listener-uri is a jar file within the ear which contains the
implementation.

         If the listener-uri is not specified, then its assumed that the class is

          visible to the application.


          Example:

            <listener>

              <listener-class>mypackage.MyClass</listener-class>

              <listener-uri>myjar.jar</listener-uri>

            </listener>


          -->

<complexType name="listenerType">

<sequence>

<element name="listener-class" type="string"/>

<element name="listener-uri" type="string" minOccurs="0"/>

</sequence>

</complexType>

<!--
```

```
          The startup element is used to register user defined startup classes.
          The startup-class element is used to define the name of the class
          intended to be run when the application is being deployed. The
          startup-uri element is used to define a jar file within the ear which
          contains the startup-class. If startup-uri is not defined, then
          its assumed that the class is visible to the application.

          Startup Listeners are deprecated in WebLogic Server 9.0.0.
          ApplicationLifecycleListeners should be used instead.

          Example:
            <startup>
              <startup-class>mypackage.MyClass</startup-class>
              <startup-uri>myjar.jar</startup-uri>
            </startup>

          -->
<complexType name="startupType">
<sequence>
<element name="startup-class" type="string"/>
<element name="startup-uri" type="string" minOccurs="0"/>
</sequence>
</complexType>
<!--
         The shutdown element is used to register user defined shutdown classes.
          The shutdown-class element is used to define the name of the class
          intended to be run when the application is being undeployed. The
```

```
          shutdown-uri element is used to define a jar file within the ear which
           contains the shutdown-class. If shutdown-uri is not defined, then
           its assumed that the class is visible to the application.

           Shutdown Listeners are deprecated in WebLogic Server 9.0.0.
           ApplicationLifecycleListeners should be used instead.

           Example:
             <shutdown>
               <shutdown-class>mypackage.MyClass</shutdown-class>
               <shutdown-uri>myjar.jar</shutdown-uri>
             </shutdown>

           -->
<complexType name="shutdownType">
<sequence>
<element name="shutdown-class" type="string"/>
<element name="shutdown-uri" type="string" minOccurs="0"/>
</sequence>
</complexType>
<!--
        The module element represents a single WebLogic application module and
contains a
        jdbc or jms element.
        -->
<complexType name="weblogic-moduleType">
<sequence>
<element name="name" type="string"/>
```

```
<element name="type">
<simpleType>
<restriction base="string">
<enumeration value="JMS"/>
<enumeration value="JDBC"/>
<enumeration value="Interception"/>
</restriction>
</simpleType>
</element>
<element name="path" type="string"/>
</sequence>
</complexType>
<!--
     This element references a J2EE library that is required by the application.

      Example:

           <library-ref>
             <library-name>WorkFlowEJB</library-name>
             <specification-version>8.1</specification-version>
             <implementation-version>0.1</implementation-version>
             <exact-match>false</exact-match>
           </library-ref>
  -->
<complexType name="library-refType">
<sequence>
<element name="library-name" type="string" minOccurs="1" maxOccurs="1"/>
```

```
<element name="specification-version" type="string" minOccurs="0"
maxOccurs="1"/>

<element name="implementation-version" type="string" minOccurs="0"
maxOccurs="1"/>

<element name="exact-match" type="boolean" minOccurs="0" maxOccurs="1"/>

<element name="context-root" type="string" minOccurs="0" maxOccurs="1"/>

<!--

        The name of the referenced library (required).

      -->

<!--

        The minimum specification-version required (optional).

      -->

      <!--

        The minimum implementation-version requited (optional).

      -->

<!--

        Controls "match highest version" behavior. Default is false (optional).

      -->

      <!--

        For webapp libraries, allows setting their context-root (optional).

      -->

</sequence>

</complexType>

</schema>
```

APPENDIX B

weblogic.Configure Command Line Reference

`weblogic.Configure` is a Java-based deployment configuration tool. It is primarily intended for developers who want to export portions of a WebLogic Server deployment configuration into an XML deployment plan. `weblogic.Configure` also provides basic JSR-88 deployment configuration functionality, enabling you to generate a basic WebLogic Server configuration for applications that have only J2EE deployment descriptors.

See also WebLogic Scripting Tool for information about performing deployment configuration operations using the WebLogic Scripting Tool (WLST).

The following sections describe using the `weblogic.Configure` utility:

Overview of weblogic.Configure

`weblogic.Configure` generates WebLogic Server deployment configuration files for an application or standalone module. `weblogic.Configure` provides two primary functions:

- Exporting different categories WebLogic Server deployment descriptor properties into empty (null) variables in a template deployment plan. You can optionally use an existing deployment plan as input to the configuration export session. Template plans are generally modified further before they can be used.
- Generating a simple deployment plan by prompting the user for values to assign to WebLogic Server descriptor properties. This function models a basic JSR-88 configuration session, where J2EE descriptors are evaluated associated WebLogic Server descriptor properties defined.

Required Environment for weblogic.Configure

To set up your environment to use the `weblogic.Configure` utility:

1. Install and configure the WebLogic Server software, as described in the WebLogic Server Installation Guide.
2. Add the WebLogic Server classes to the `CLASSPATH` environment variable, and ensure that the correct JDK binaries are available in your `PATH`. You can use the `setWLSEnv.sh` or `setWLSEnv.cmd` script, located in the `server/bin` subdirectory of the WebLogic Server installation directory, to set the environment.

Syntax for Invoking weblogic.Configure

```
java weblogic.Configure [Options] [filespec]
```

The *`filespec`* can be either:

- An absolute or relative path to an archive file
- An absolute or relative path to an exploded archive directory

If you want to specify an application installation root directory, use the -root option as described in "Options" on page B-2. In all cases, the application identified with the *`filespec`* must contain valid J2EE deployment descriptor files.

Options

The following table describe each `weblogic.Configure` option.

Figure 0-1 weblogic.Configure Options

Option	Description
`-debug`	Enables debug mode.
`-export [all | dependencies | declarations | dynamics]`	Generates null variable definitions in a template deployment plan for different categories of deployment configuration property: • `all`—Generates null variables for all WebLogic Server deployment descriptor properties that can be configured by an Administrator. • `dependencies`—Generates null variables for all WebLogic Server descriptor properties that resolve external resource references. • `declarations`—Generates null variables for all properties that declare a resource to other applications and modules. • `dynamics`—Generates variables for all properties that can be changed on-the-fly without requiring the application to be redeployed. If you omit the `-export` option, `weblogic.Configure` performs a basic JSR-88 configuration session and prompts you for WebLogic Server descriptor property values associated with the application's J2EE configuration.
`-noprompt`	Disables prompting for user input when generating a deployment plan.
`-plan` *`plan_file`*	Identifies the path and name of the plan file to create for the configuration session. By default, the plan is written to `/plan/plan.xml` in the application's root directory.
`-root` *`root_directory`*	Specifies an application root directory on which to perform the configuration or export. If you do not specify a path, `weblogic.Configure` searches in the default path of *`TEMP`*`/weblogic-install/`*`application`* where *`TEMP`* is the temporary directory for your environment and *`application`* is the name of the application. For example, under Windows the default root path for an application named `myapp` is `c:\Documents and Settings\`*`username`*`\Local Settings\Temp\myapp`.

Option	Description
`-type [ear | war | jar | car | rar | jms | jdbc]`	Identifies the type of application or module you have selected for configuration. If the `-type` option is omitted, `weblogic.Deployet` attempts to determine the application type from a filename extension, if available. File names using a `.jar` extension are treated as EJB modules; if you are configuring a client JAR file, you must manually specify the correct `-type`. JMS and JDBC modules both share the `.xml` extension. However, `weblogic.Deployer` determines the exact type using the file contents.
`-useplan plan_file`	Specifies an existing deployment plan file to use as input to an export session. If you use `-root` to specify an application root directory, `weblogic.Configure` uses the `/plan/plan.xml` file in the root directory as input, if one is available.
`-variables [global | unique]`	Specifies whether variable names created in the deployment plan can be used across all modules of an application, or only within a particular module. For example, a role assignment might be applied to an entire EAR, or to only a single Web Application or other module within the EAR. By default, `Configure` creates global variables that apply to the entire application.

weblogic.Configure Examples

The following sections describe common configuration and export scenarios, with examples examples of `weblogic.Configure` syntax.

Configuring a Basic J2EE Application

The following command creates a deployment plan for an archived Enterprise Application by prompting the user for individual WebLogic Server deployment property values:

```
java weblogic.Configure -plan plan.xml ./myApplication.ear
```

By default, the WebLogic Server deployment API creates new application root directories in the *`TEMP`*`/weblogic-install/application_name` subdirectory, where *`TEMP`* is the temporary directory for your environment and *`application_name`* archive filename. In this example,

generated deployment files would be created in `C:\Documents and Settings\username\Local Settings\Temp\weblogic\install\myApplication.ear\config`.

Configuring an Application in A Root Directory

If you store your application using an installation root directory, generated configuration files are automatically stored in the `/plan` subdirectory of the root:

```
java weblogic.Configure -root /appRelease/MyApplication
```

In the above example, the `plan.xml` file and any generated WebLogic Server deployment descriptors are automatically stored in `/appRelease/MyApplication/plan`.

Configuring Using an Existing Deployment Plan

The following command uses an existing plan as input to the configuration session, and generates a new plan in the `/plan` subdirectory of the application root directory:

```
java weblogic.Configure -useplan /plans/MyApplication_template.xml
     -root /appRelease/MyApplication
```

Exporting a WebLogic Server Deployment Configuration

The `-export` command creates null variables for a subset of WebLogic Server deployment descriptor properties relevant to the application or module. The following plan exports all configurable properties to null variables in a template deployment plan:

```
java weblogic.Configure -export all -root /appRelease/MyApplication
```

See "Exporting an Application for Deployment to New Environments" on page 8-1 for more information about exporting a deployment configuration.

APPENDIX C

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 C-3 and "wldeploy Ant Task Attribute Reference" on page C-3.
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 output shows a `wldeploy` target that deploys an application to a single WebLogic Server instance:

```
<target name="deploy">
   <wldeploy action="deploy"
      source="${build}/ejb11_basic_statelessSession.ear" name="ejbapp"
      user="a" password="a" verbose="true" adminurl="t3://localhost:7001"
      debug="true" targets="myserver"/>
</target>
```

wldeploy Ant Task Attribute Reference

The following table describes the attributes of the `wldeploy` Ant task. For more information about the definition of various terms, see the weblogic.Deployer Command-Line Reference in *Deploying Applications to WebLogic Server*.

Table 0-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type	Required?
action	The deployment action to perform. Valid values are `deploy`, `cancel`, `undeploy`, `redeploy`, `distribute`, `start`, and `stop`.	String	No
adminurl	The URL of the Administration Server.	String	No
altappdd	Specifies the name of an alternate J2EE deployment descriptor (`application.xml`) to use for deployment.	String	No

Table 0-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type	Required?
altwlsappdd	Specifies the name of an alternate WebLogic Server deployment descriptor (`weblogic-application.xml`) to use for deployment.	String	No
debug	Enable `wldeploy` debugging messages.	boolean	No
external_stage	Specifies whether the deployment uses external_stage deployment mode.	boolean	No
files	Specifies a list of files on which to perform a deployment action (for example, a list of JSPs to redeploy).	FileSet	No
id	Identification used for obtaining status or cancelling the deployment.	String	No
name	The deployment name for the deployed application.	String	No
nostage	Specifies whether the deployment uses nostage deployment mode.	boolean	No
nowait	Specifies whether wldeploy returns immediately after making a deployment call (by deploying as a background task).	boolean	No

Table 0-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type	Required?
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	No
remote	Specifies whether the server is located on a different machine. This affects how filenames are transmitted.	boolean	No
source	The source file to deploy.	File	No
stage	Specifies whether the deployment uses stage deployment mode.	boolean	No
targets	The list of target servers to deploy to.	String	No
timeout	The maximum time to wait for a deployment to succeed.	int	No
upload	Specifies whether the source file(s) are copied to the Administration Server's upload directory prior to deployment.	boolean	No
user	The administrative username.	String	No

Table 0-1 Attributes of the wldeploy 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 `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	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.Admin Command-Line Reference*.	String	No
verbose	Specifies whether `wldeploy` displays verbose output messages.	boolean	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

Index

M

P

R

S

T

W