BEA WebLogic Server™

Developing WebLogic Server Applications

Version 8.1
Revised: September 23, 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

1. Understanding WebLogic Server Applications and Basic Concepts

2. Creating WebLogic Server Applications

3. Programming Topics

4. WebLogic Server Application Classloading

A. Enterprise Application Deployment Descriptor Elements

B. Client Application Deployment Descriptor Elements

About This Document

This document introduces the BEA WebLogic Server™ application development environment. It describes how to establish a development environment and how to package applications for deployment on the WebLogic Server platform.

The document is organized as follows:

- Chapter 1, "Understanding WebLogic Server Applications and Basic Concepts," describes modules of WebLogic Server applications.
- Chapter 2, "Creating WebLogic Server Applications," introduces the split development directory structure and outlines the steps involved in creating Enterprise applications.
- Chapter 3, "Programming Topics," covers general WebLogic Server application programming issues, such as logging messages and using threads.
- Chapter 4, "WebLogic Server Application Classloading," provides an overview of Java classloaders, followed by details about WebLogic Server application classloading.
- Appendix A, "Enterprise Application Deployment Descriptor Elements," is a reference for the standard J2EE Enterprise application deployment descriptor, `application.xml` and the WebLogic-specific application deployment descriptor `weblogic-application.xml`.
- Appendix B, "Client Application Deployment Descriptor Elements," is a reference for the standard J2EE Client application deployment descriptor, `application-client.xml`, and the WebLogic-specific client application deployment descriptor.

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.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. The following WebLogic Server documents contain information that is relevant to creating WebLogic Server application modules:

- *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81/ejb/index.html
- *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs81/servlet/index.html

- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81/jsp/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs81/webapp/index.html
- *Programming WebLogic JDBC* at http://e-docs.bea.com/wls/docs81/jdbc/index.html
- *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs81/webServices/index.html
- *Programming WebLogic J2EE Connectors* at http://e-docs.bea.com/wls/docs81/jconnector/index.html

For more information in general about Java application development, refer to the Sun Microsystems, Inc. Java 2, Enterprise Edition Web Site at http://java.sun.com/products/j2ee/.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`samples/domains/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Example*s: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.Deployer [list|deploy|undeploy|update] password {application} {source}`

Convention	Usage
. . .	Indicates one of the following in a command line: • An argument can be repeated several times in the command line. • The statement omits additional optional arguments. • You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER 1

Understanding WebLogic Server Applications and Basic Concepts

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

J2EE Platform and WebLogic Server

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

What Are WebLogic Server 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 "Using Ant Tasks to Create Compile Scripts" on page 3-4.

You can also precompile JSPs and package the servlet class in a Web archive (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:

- "Best Practices for Developing WebLogic Server Applications" on page 2-22.

- "Split Development Directory Ant Tasks" on page 2-4.
- *Developing Web Applications for WebLogic Server*
- *Programming WebLogic Server HTTP Servlets*
- *Programming WebLogic JSP*
- *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 "Compiling Java Code" on page 3-2.

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 J2EE Connectors.*

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," and Appendix B, "Client Application Deployment Descriptor Elements."

For both production and development purposes, BEA recommends that you package and deploy even stand-alone Web applicatons, 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 faciliates application development. See "Overview of the Split Development Directory Environment" on page 2-2.

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 "Overview of the Split Development Directory Environment" on page 2-2. 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 2-18.
- 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 2-6.
- 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 "Organizing J2EE Components in a Split Development Directory" on page 2-6.

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. They are implemented using standard J2EE modules, such as Enterprise Java Beans, or with a Java class. They are packaged as standard J2EE Enterprise applications that contain a Web Application (which contains the Web Service deployment descriptor file and the class files for Java class-implemented Web Services) and the EJB JAR file for EJB-implemented Web Services.
- 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).

Client Applications

Java clients that access WebLogic Server application modules range from simple command line utilities that use standard I/O to highly interactive GUI applications built using the Java Swing/AWT classes. Java clients access WebLogic Server modules indirectly through HTTP requests or RMI requests. The modules execute requests in WebLogic Server, not in the client.

In previous versions of WebLogic Server, a Java client required the full WebLogic Server JAR on the client machine. WebLogic Server 8.1 supports a true J2EE application client, referred to as the *thin client*. A small footprint standard JAR and a JMS JAR—`wlclient.jar` and `wljmsclient.jar` respectively—are provided in the `/server/lib` subdirectory of the WebLogic Server installation directory. Each JAR file is about 400 KB.

A J2EE application client runs on a client machine and can provide a richer user interface than can be provided by a markup language. Application clients directly access Enterprise JavaBeans running in the business tier, and may, as appropriate communicate through HTTP with servlets running in the Web tier. Although a J2EE application client is a Java application, it differs from a stand-alone Java application client because it is a J2EE module, hence it offers the advantages of portability to other J2EE-compliant servers, and can access J2EE services. For more information about the thin client, see "Developing a J2EE Application Client (Thin Client)" in *Programming WebLogic RMI over IIOP*.

For more information about all client types supported by WebLogic Server, see "Overview of RMI-IIOP Programming Models" in *Programming WebLogic RMI over IIOP*.

XML Deployment Descriptors

Modules and applications have deployment descriptors—XML documents—that describe the contents of the directory or JAR file. Deployment descriptors are text documents formatted with XML tags. The J2EE specifications define standard, portable deployment descriptors for J2EE modules and applications. BEA defines additional WebLogic-specific deployment descriptors for deploying a module or application in the WebLogic Server environment.

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

Table 1-1 J2EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Web Application	J2EE	`web.xml` See "web.xml Deployment Descriptor Elements" in *Developing Web Applications for WebLogic Server*.
	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 Deployment Descriptor Elements" in *Programming WebLogic Server J2EE Connectors*.
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-6.

Table 1-1 J2EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Client Application	J2EE	`application-client.xml` See "application-client.xml Deployment Descriptor Elements" on page B-2.
	WebLogic	`client-application.runtime.xml` See "WebLogic Run-time Client Application Deployment Descriptor" on page B-5.

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. You can use a variety of tools to do this. See "Editing Deployment Descriptors" on page 1-11.

Automatically Generating Deployment Descriptors

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

WebLogic Builder

WebLogic Builder is a WebLogic Server tool for generating and editing deployment descriptors for WebLogic Server applications. It can also deploy WebLogic Server applications to single servers. See "Deployment Tools Reference" in *Deploying WebLogic Server Applications*.

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.

Editing Deployment Descriptors

BEA offers a variety of tools for editing the deployment descriptors of WebLogic Server applications and modules. Using these tools, you can update existing elements in, add new elements to, and delete existing elements from deployment descriptors. These tools include:

- WebLogic Builder—WebLogic Server tool for generating and editing deployment descriptors for WebLogic Server applications. It can also deploy WebLogic Server applications to single servers. See "Deployment Tools Reference" in *Deploying WebLogic Server Applications*.
- XML Editor with DTD validation—Such as BEA XML Editor on dev2dev or XMLSpy. (An evaluation copy of XMLSpy is bundled with this version of WebLogic Server.) See *BEA dev2dev Online* at http://dev2dev.bea.com/index.jsp.

- Administration Console Descriptor tab—This release of WebLogic Server has deprecated the Deployment Descriptor Editor. A new Administration Console Descriptor tab in the has replaced it. Use the Descriptor tab to view, modify, and persist deployment descriptor elements to the descriptor file within WebLogic Server applications in the same manner that they were persisted using the Deployment Descriptor Editor. However, these descriptor element changes take place dynamically at runtime without requiring the resource adapter to be redeployed. The descriptor elements in the Descriptor tab include only those descriptor elements that can be dynamically changed at runtime. See the Administration Console Online Help.

Development Software

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

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

Creating WebLogic Server Applications

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

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 2-6.

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)

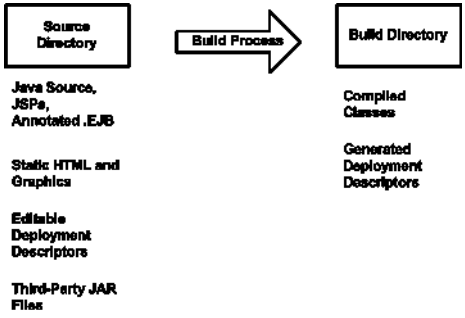
  Note: The split development directory structure does not provide support for developing new Resource Adapter components.

- 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 2-1 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,

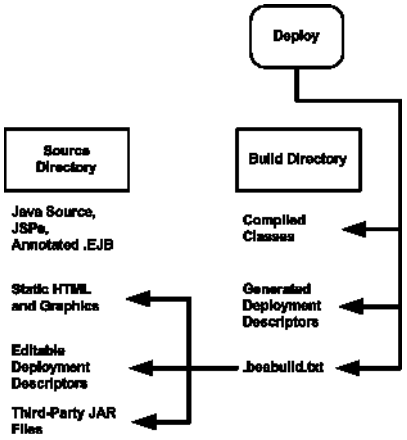
WebLogic Server then looks in the application's build directory for that resource. For example, if a deployment descriptor is generated during the build process, rather than stored with source code as an editable file, WebLogic Server obtains the generated file from the build directory.

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 Applications Using wldeploy" on page 2-17 describes the `wldeploy` Ant task that you can use to automate deployment from the split directory environment.

Figure 2-2, "Split Directory Deployment," on page 2-4 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 2-2 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 2-16.
- `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 2-16.
- `wlappc`—This Ant task invokes the appc compiler, which generates JSPs and container-specific EJB classes for deployment. See "appc Compiler" on page 3-2.
- `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 "Deployment Tools Reference" in *Deploying WebLogic Server Applications*.
- `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 2-6.
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 2-6 and "Organizing Shared Classes in a Split Development Directory" on page 2-12.
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 2-13.
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 2-13 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 2-19.

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

- "Enterprise Application Configuration" on page 2-9
- "Web Applications" on page 2-9
- "EJBs" on page 2-11
- "Shared Utility Classes" on page 2-12
- "Third-Party Libraries" on page 2-13

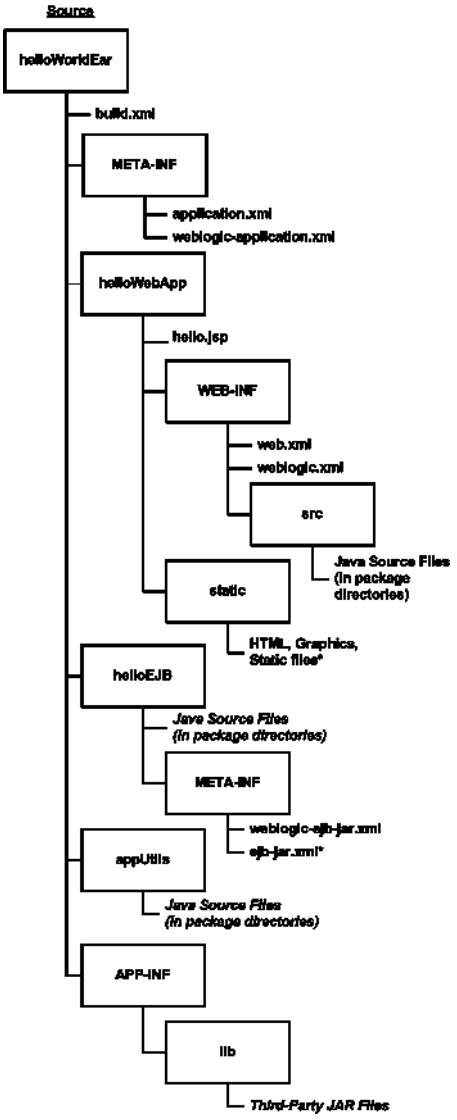
Note: WebLogic Server does not provide additional support for developing J2EE Connectors using the split development directory.

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.

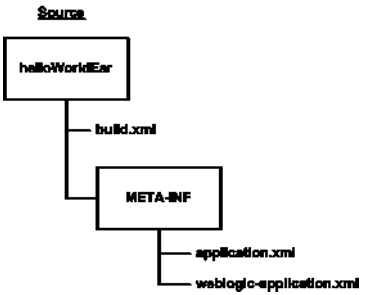
Figure 2-3 Overview of Enterprise Application Source Directory



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 2-4 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:

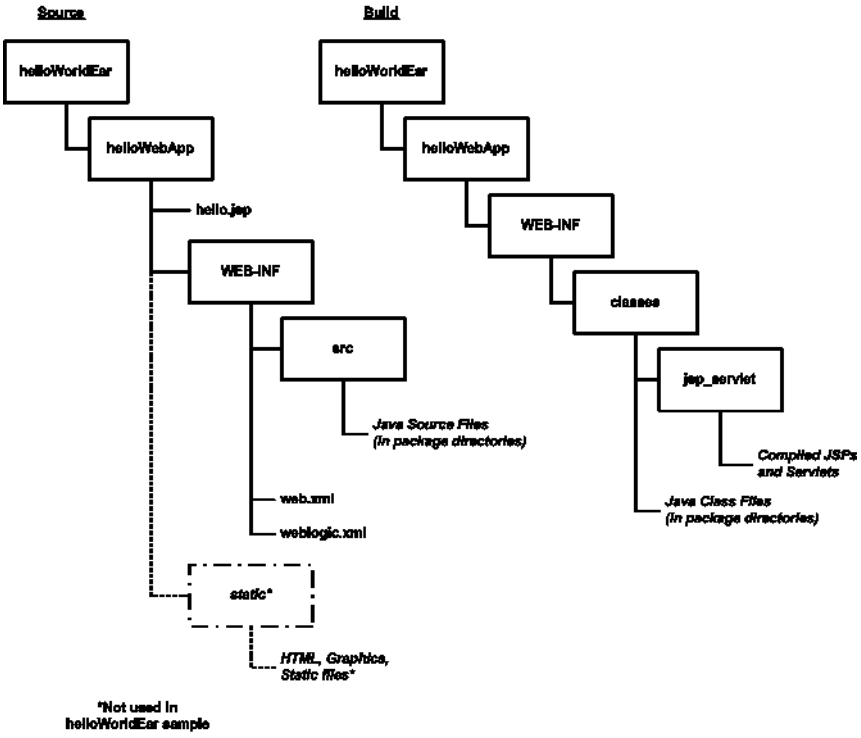
```
java weblogic.marathon.ddinit.EarInit \myEAR
```

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 2-5 Web Application Source and Build Directories



The key directories and files for the Web Application are:

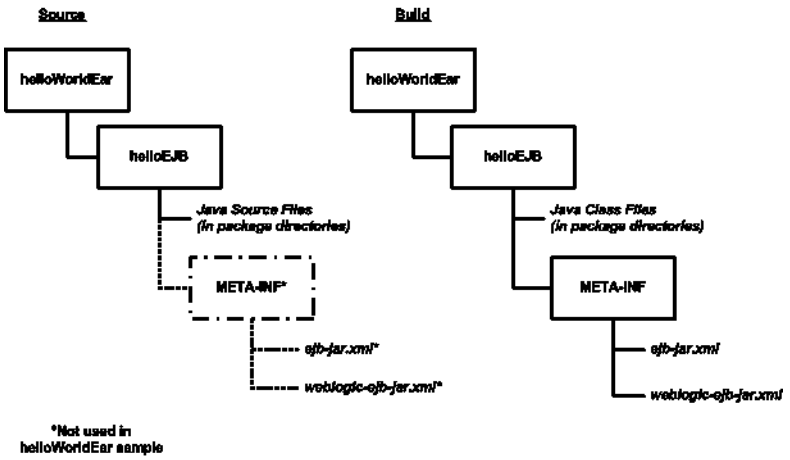
- `helloWebApp\` —The top level of the Web Application module can contain JSP files and static content such as HTML files and graphics used in the application. You can also store static files in any named subdirectory of the Web Application (for example, `helloWebApp\graphics` or `helloWebApp\static`.)
- `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 2-6 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 2-11.

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.

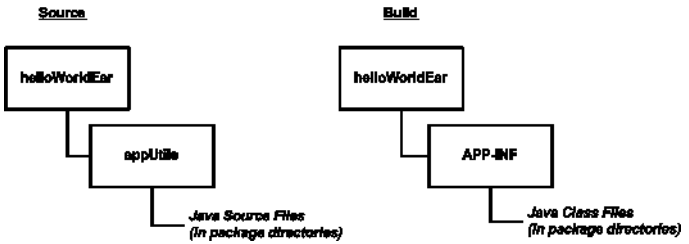
Organizing Shared Classes in a Split Development Directory

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 2-7 Java Utility Class Directory



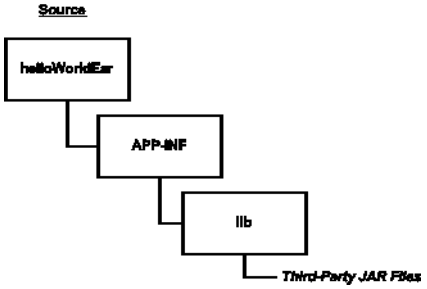
Place the source for Java utility classes in a named subdirectory of the top-level Enterprise Application directory. Beneath the named subdirectory, use standard package subdirectory conventions.

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 2-8 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
- `-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 2-1 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 2-2 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
```

Generating Deployment Descriptors Using wlddcreate

The `wlddcreate` ant task is provided as 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 Options

Table 2-1 contains Ant task options specific to `wlcompile`.

Table 2-1 wlcompile Ant Task Options

Option	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.

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>
```

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 in *Deploying WebLogic Server Applications* 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 Exploded Archive Directories in *Deploying WebLogic Server Applications*.

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. See "Archive versus Exploded Archive Directory" on page 2-18.

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" />
```

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. See also the *Avitek Medical Records Development Tutorials* for information about the MedREc directory layout and build process.

Organizing Libraries and Classes Shared by Multiple EARs

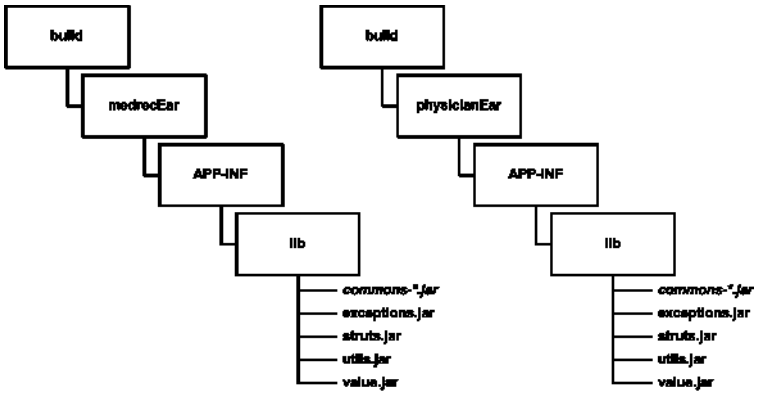
For single EAR projects, the split development directory conventions suggest keeping third-party JAR files in the `APP-INF/lib` directory of the EAR source directory. However, a multiple-EAR project would require you to maintain a copy of the same third-party JAR files in the `APP-INF/lib` directory 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 2-9 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 2-9.

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. Also, see the various "Best Practices" sections in the *MedRec Tutorials*.

- Package applications as part of an Enterprise application. See "Packaging Applications Using wlpackage" on page 2-18.
- Use the split development directory structure. See "Organizing J2EE Components in a Split Development Directory" on page 2-6.
- For distribution purposes, package and deploy in archived format. See "Packaging Applications Using wlpackage" on page 2-18.
- In most other cases, it is more convenient to deploy in exploded format. See "Archive versus Exploded Archive Directory" on page 2-18.
- 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 3

Programming Topics

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

Compiling Java Code

In general, compilers convert information from one form to another. Traditionally, this involves converting information from user readable source to machine readable form. For example, the `javac` compiler converts `.java` files to `.class` files while the appc compiler generates EJBs and JSPs for deployment.

You can also use Ant tasks, which automate the various compile operations that must be performed to build an application, making it simpler to compile (or build) WebLogic Server-specific applications, especially in a development environment. See "Using Ant Tasks to Create Compile Scripts" on page 3-4.

javac Compiler

The Sun Microsystems `javac` compiler reads class and interface definitions, written in the Java programming language and compiles them into the bytecode class files. See "javac - Java Programming Language Compiler."

BEA provides the `wlcompile` Ant task to invoke the `javac` compiler. See "wlcompile Ant Task" on page 3-5.

appc Compiler

To generate JSPs and container-specific EJB classes for deployment, you use the `weblogic.appc` compiler. The `appc` compiler also validates the descriptors for compliance with the current specifications at both the individual module level and the application level. The application-level checks include checks between the application-level deployment descriptors and the individual modules as well as validation checks across the modules.

The `appc` 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.

appc Syntax

Use the following syntax to run `appc`:

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

appc Options

The following are the available `appc` options:

Table 3-1 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 will not be regenerated unless a checksum indicates that it is necessary.
`-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.

Using Ant Tasks to Create Compile Scripts

The preferred BEA method for compiling 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. 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 `samples\domains\examples` directory.

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:

```
samples\domains\examples\ejb20\basic\beanManaged\build.xml
```

Also refer to the following WebLogic Server documentation on building examples using Ant

```
samples\server\examples\src\examples\examples.html
```

wlcompile Ant Task

You use the `wlcompile` Ant task to call the `javac` compiler to compile your Enterprise application's Java files in a split development directory structure. For more information, refer to "Compiling Applications Using wlcompile" on page 2-16 in Chapter 2, "Creating WebLogic Server Applications."

wlappc Ant Task

Use the `wlappc` Ant task to invoke the `appc` compiler, which generates JSPs and container-specific EJB classes for deployment.

wlappc Ant Task Options

Table 3-2 contains Ant task options specific to `wlappc`. For the most part, these options are the same as `weblogic.appc` options. However, there are a few differences.

Note: See "appc Compiler" on page 3-2 for a list of `weblogic.appc` options.

Table 3-2 wlappc Ant Task 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 will not be regenerated unless a checksum indicates that it is necessary.
`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.
`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 3-2.

```
<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"/>
```

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.

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.

If you create an InitialContext in the threads, ensure that you explicitly close `InitialContext` to release resources immediately and avoid any potential memory leaks. See Closing the Context in *Programming WebLogic JNDI* for more information.

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.

About 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.

`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.

Property	Description	Default
`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 3-12. 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.

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 "Increased Reliability and Availability for Clustered EJBs" 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.

CHAPTER 4

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 4-3section 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 4-1 illustrates the `prefer-web-inf-classes` element, its description and default value.

Listing 4-1 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 4-4 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 4-15.

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.

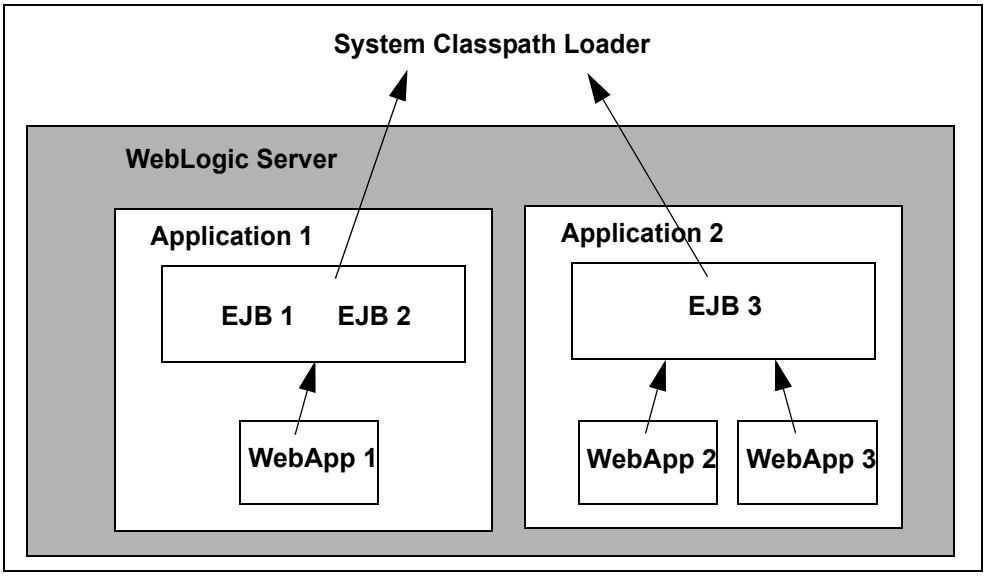
Application Classloader Hierarchy

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 4-1 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 4-14.

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.

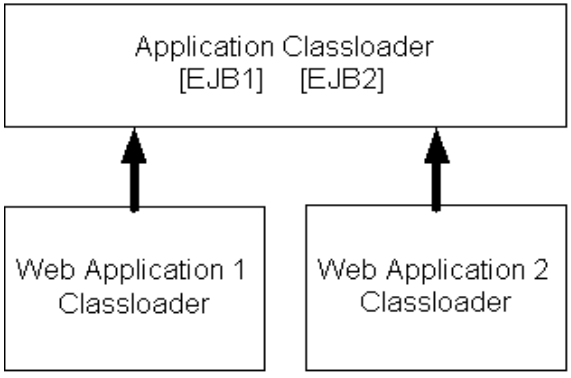
Custom Module Classloader Hierarchies

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 4-2 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`. For instructions on how to edit deployment descriptors, refer to the "WebLogic Builder Online Help."

The DTD for this declaration is as follows:

Listing 4-2 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 4-3 Example Classloader Declaration

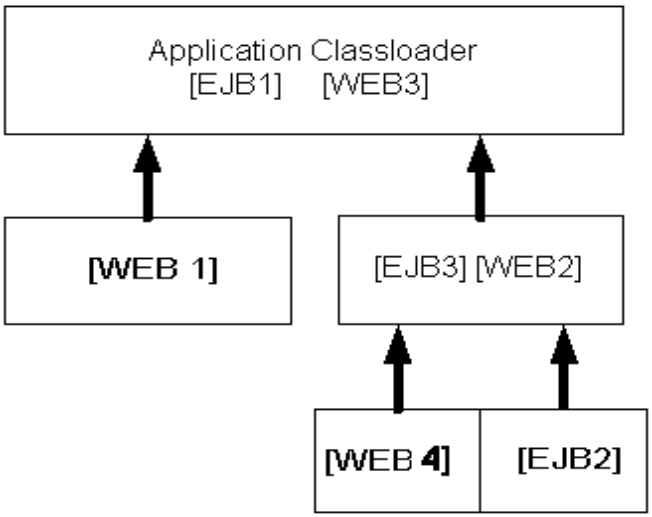
```
<classloader-structure>
        <module-ref>
                <module-uri>ejb1.jar</module-uri>
        </module-ref>
        <module-ref>
                <module-uri>web3.war</module-uri>
        </module-ref>

        <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 4-3 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.

Web Applications prefer-web-inf-classes Flag Ignored

If you use a custom classloader hierarchy, the `prefer-web-inf-classes` flag will be ignored for web applications within that hierarchy.

Custom Classloader Structure with Iterative Development

When a new classloader-structure element is added as a leaf node anywhere in the existing class-loader hierarchy, then the module added to the new classloader-structure can be deployed without redeploying the entire application. However, when deleting or rearranging the existing classloader-structure element within the hierarchy, the entire application should be redeployed.

When you add new module-uri(s) to an existing classloader-stucture, ensure that it is added only after the existing module-uri(s). New module(s) can be deployed without redeploying the entire application. However, when moving or deleting module-uri(s) across classloader-stucture elements, ensure that you redeploy the entire 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 occuring while a change is being made) might end up accessing different versions of classes depending on timing.

Development Use Only

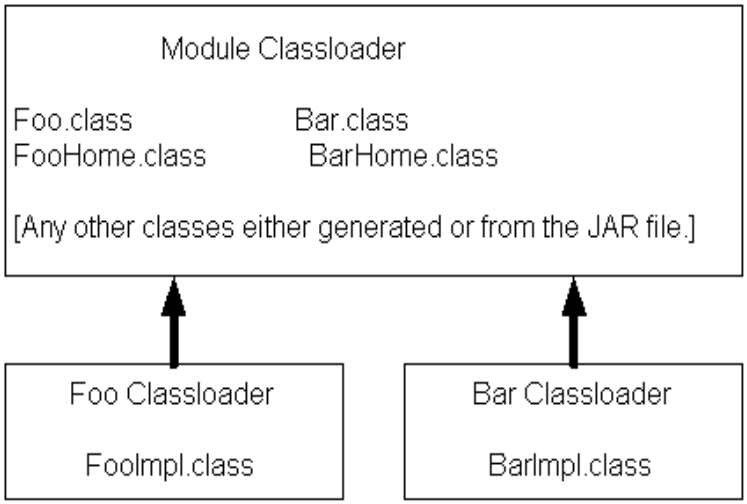
The development-use-only feature is intended for development use. Because updates are not atomic, this feature is not suitable for production use.

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 4-4 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 4-4 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 4-5 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.

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

The following sections describe the `application.xml` file. The `application.xml` file is the deployment descriptor for an Enterprise application. The file is located in the `META-INF` subdirectory of the application archive. It must begin with the following DOCTYPE declaration:

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
```

application

The `application` element is the root element of the application deployment descriptor. The elements within the `application` element are described in the following sections.

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

Element	Required Optional	Description
`<icon>`	Optional	Specifies the locations of small and large images that represent the application in a GUI tool. This element is not currently used by WebLogic Server. For more information on the elements you can define within the `icon` element, refer to "icon" on page A-3.
`<display-name>`	Required	Specifies the application display name, a short name that is intended to be displayed by GUI tools.
`<description>`	Optional	Provides descriptive text about the application.

Element	Required Optional	Description
<module>	Required	The application.xml deployment descriptor contains one module element for each module within the Enterprise application. Each module element contains an connector, ejb, java, or web element that indicates the module type and location of the module within the application. An optional alt-dd element specifies an optional URI to the post-assembly version of the deployment descriptor. For more information on the elements you can define within the module element, refer to "module" on page A-4.
<security-role>	Required	Contains the definition of a security role which is global to the application. Each security-role element contains an optional description element, and a role-name element. For more information on the elements you can define within the security-role element, refer to "security-role" on page A-6.

icon

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

Element	Required Optional	Description
<small-icon>	Optional	Specifies the location for a small (16x16 pixel) .gif or .jpg image used to represent the application in a GUI tool. Currently, this is not used by WebLogic Server.
<large-icon>	Optional	Specifies the location for a large (32x32 pixel) .gif or .jpg image used to represent the application in a GUI tool. Currently, this element is not used by WebLogic Server.

module

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

Element	Required Optional	Description
`<alt-dd>`	Optional	Specifies an optional URI to the post-assembly version of the deployment descriptor file for a particular J2EE module. The URI must specify the full pathname of the deployment descriptor file relative to the application's root directory. If you do not specify `alt-dd`, the deployer must read the deployment descriptor from the default location and file name required by the respective module specification. You can specify an alternate deployment descriptor only for the J2EE deployment descriptors, `web.xml` and `ejb-jar.xml`. You cannot specify alternate descriptor files for the `weblogic.xml` or `weblogic-ejb-jar.xml`.
`<connector>`	Required	Specifies the URI of a resource adapter (connector) archive file, relative to the top level of the application package.
`<ejb>`	Required	Defines an EJB module in the application file. Contains the path to an EJB JAR file in the application. Example: `<ejb>petStore_EJB.jar</ejb>`

Element	Required Optional	Description
<java>	Required	Defines a client application module in the application file. Example: <java>client_app.jar</java>
<web>	Required	Defines a Web application module in the application.xml file. The web element contains a web-uri element and a context-root element. If you do not declare a value for the context-root, then the basename of the web-uri element is used as the context path of the Web application. (Note that the context path must be unique in a given Web server. More than one Web application may be using the same Web server, so you must avoid context path clashes across multiple applications.) **web-uri** Defines the location of a Web module in the application.xml file. This is the name of the WAR file. **context-root** Specifies a context root for the Web application. Example: <web> <web-uri>petStore.war</web-uri> <context-root>estore</context-root> </web>

security-role

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

Element	Required Optional	Description
`<description>`	Optional	Text description of the security role.

`<role-name>`	Optional	Defines the name of a security role or principal that is used for authorization within the application. Roles are mapped to WebLogic Server users or groups in the `weblogic-application.xml` deployment descriptor.
		Example:
		`<security-role>`
		`  <description>the gold customer role</description>`
		`  <role-name>gold_customer</role-name>`
		`</security-role>`
		`<security-role>`
		`  <description>the customer role</description>`
		`  <role-name>customer</role-name>`
		`</security-role>`

weblogic-application.xml Deployment Descriptor Elements

The following sections describe the `weblogic-application.xml` file. 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 pools and EJB caching.

The file is located in the `META-INF` subdirectory of the application archive. It must begin with the following DOCTYPE declaration:

```
<!DOCTYPE weblogic-application PUBLIC "-//BEA Systems, Inc.//DTD
WebLogic Application 8.1.0//EN"
"http://www.bea.com/servers/wls810/dtd/weblogic-application_2_0.dtd">
```

The following sections describe each element 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-10.
`<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-13.
`<jdbc-connection -pool>`		**Deprecated** 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-15.
`<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-27.

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 and for responses. • webapp.encoding.default—Can be set to a string representing an encoding supported by the JDK. If set, this defines the default encoding used to process servlet requests and servlet responses. This setting is ignored if webapp.encoding.usevmdefault is set to true. This value is also overridden for request streams by the input-charset element of weblogic.xml. • webapp.encoding.usevmdefault—Can be set to true or false. If true, the system property file.encoding is used to define the default encoding. The following parameter is used to affect the behavior of Web applications that are contained in this application. • webapp.getrealpath.accept_context_path—This is a compatibility switch that may be set to true or false. If set to true, the context path of Web applications is allowed in calls to the servlet API getRealPath. Example: <application-param> <param-name> webapp.encoding.default </param-name> <param-value>UTF8</param-value> </application-param> For more information on the elements you can define within the application-param element, refer to "application-param" on page A-28.

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-28.

`<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-28.
`<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-29.
`<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-29.

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-11.

`<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-13.
`<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-14.

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-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-16.
`<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-17.
`<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-24.

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	

Element	Required Optional	Description
<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.

APPENDIX B

Client Application Deployment Descriptor Elements

The following sections describe deployment descriptors for J2EE client applications on WebLogic Server. Often, when it comes to J2EE applications, users are only concerned with the server-side modules (Web applications, EJBs, connectors). You configure these server-side modules using the `application.xml` deployment descriptor, discussed in Appendix A, "Enterprise Application Deployment Descriptor Elements."

However, it is also possible to include a client module (a JAR file) in an EAR file. This JAR file is only used on the client side; you configure this client module using the `client-application.xml` deployment descriptor. This scheme makes it possible to package both client and server side modules together. The server looks only at the parts it is interested in (based on the `application.xml` file) and the client looks only at the parts it is interested in (based on the `client-application.xml` file).

For client-side modules, two deployment descriptors are required: a J2EE standard deployment descriptor, `application-client.xml`, and a WebLogic-specific runtime deployment descriptor with a name derived from the client application JAR file.

application-client.xml Deployment Descriptor Elements

The `application-client.xml` file is the deployment descriptor for J2EE client applications. It must begin with the following DOCTYPE declaration:

```
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application Client 1.2//EN"
"http://java.sun.com/j2ee/dtds/application-client_1_2.dtd">
```

The following sections describe each of the elements that can appear in the file.

application-client

`application-client` is the root element of the application client deployment descriptor. The application client deployment descriptor describes the EJB modules and other resources used by the client application.

The following table describes the elements you can define within an `application-client` element.

Element	Required Optional	Description
`<icon>`	Optional	Specifies the locations of small and large images that represent the application in a GUI tool. This element is not currently used by WebLogic Server.
`<display-name>`		Specifies the application display name, a short name that is intended to be displayed by GUI tools.
`<description>`	Optional	The `description` element provides a description of the client application.

Element	Required Optional	Description
`<env-entry>`		Contains the declaration of a client application's environment entries. Elements that can be defined within the `env-entry` element are: • `description`—Optional. The `description` element contains a description of the particular environment entry. • `env-entry-name`—The `env-entry-name` element contains the name of a client application's environment entry. • `env-entry-type`—The `env-entry-type` element contains the fully-qualified Java type of the environment entry. The possible values are: `java.lang.Boolean`, `java.lang.String`, `java.lang.Integer`, `java.lang.Double`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Long`, and `java.lang.Float`. • `env-entry-value`—Optional. The `env-entry-value` element contains the value of a client application's environment entry. The value must be a String that is valid for the constructor of the specified `env-entry-type`.

Element	Required Optional	Description
`<ejb-ref>`		Used for the declaration of a reference to an EJB referenced in the client application. Elements that can be defined within the `ejb-ref` element are: • `description`—Optional. The `description` element provides a description of the referenced EJB. • `ejb-ref-name`—Contains the name of the referenced EJB. Typically the name is prefixed by `ejb/`, such as `ejb/Deposit`. • `ejb-ref-type`—Contains the expected type of the referenced EJB, either `Session` or `Entity`. • `home`—Contains the fully-qualified name of the referenced EJB's home interface. • `remote`—Contains the fully-qualified name of the referenced EJB's remote interface. • `ejb-link`—Specifies that an EJB reference is linked to an enterprise JavaBean in the J2EE application package. The value of the `ejb-link` element must be the name of the `ejb-name` of an EJB in the same J2EE application.

Element	Required Optional	Description
<resource-ref>		Contains a declaration of the client application's reference to an external resource. Elements that can be defined within the resource-ref element are: • description—Optional. The description element contains a description of the referenced external resource. • res-ref-name—Specifies the name of the resource factory reference name. The resource factory reference name is the name of the client application's environment entry whose value contains the JNDI name of the data source. • res-type—Specifies the type of the data source. The type is specified by the Java interface or class expected to be implemented by the data source. • res-auth—Specifies whether the EJB code signs on programmatically to the resource manager, or whether the Container will sign on to the resource manager on behalf of the EJB. In the latter case, the Container uses information that is supplied by the Deployer. The res-auth element can have one of two values: Application or Container.

WebLogic Run-time Client Application Deployment Descriptor

This XML-formatted deployment descriptor is not stored inside of the client application JAR file like other deployment descriptors, but must be in the same directory as the client application JAR file.

The file name for the deployment descriptor is the base name of the JAR file, with the extension .runtime.xml. For example, if the client application is packaged in a file named c:/applications/ClientMain.jar, the run-time deployment descriptor is in the file named c:/applications/ClientMain.runtime.xml.

application-client

The `application-client` element is the root element of a WebLogic-specific run-time client deployment descriptor. The following table describes the elements you can define within an `application-client` element.

Element	Required Optional	Description
`<env-entry>`		Specifies values for environment entries declared in the deployment descriptor. Elements that can be defined within the `env-entry` element are: • `env-entry-name`—Contains the name of an application client's environment entry. Example: <env-entry-name>EmployeeAppDB</env-entry-name> • `env-entry-value`—Contains the value of an application client's environment entry. The value must be a valid string for the constructor of the specified type, which takes a single string parameter.

Element	Required Optional	Description
<ejb-ref>		Specifies the JNDI name for a declared EJB reference in the deployment descriptor. Elements that can be defined within the ejb-ref element are: • ejb-ref-name—Contains the name of an EJB reference. The EJB reference is an entry in the application client's environment. It is recommended that name is prefixed with ejb/. Example: <ejb-ref-name>ejb/Payroll</ejb-ref-name> • jndi-name—Specifies the JNDI name for the EJB.
<resource-ref>		Declares an application client's reference to an external resource. It contains the resource factory reference name, an indication of the resource factory type expected by the application client's code, and the type of authentication (bean or container). Example: <resource-ref> <res-ref-name>EmployeeAppDB</res-ref-name> <jndi-name>enterprise/databases/HR1984</jndi-name> </resource-ref> Elements that can be defined within the resource-ref element are: • res-ref-name—Specifies the name of the resource factory reference name. The resource factory reference name is the name of the application client's environment entry whose value contains the JNDI name of the data source. • jndi-name—Specifies the JNDI name for the resource.

Index

G

I

J

M

P

R

S

T

W