BEA WebLogic Server™

Developing WebLogic Server Applications

Release 8.1
Document Revised: October 29, 2002

Copyright

Copyright © 2002 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Developing WebLogic Server Applications

Part Number	Document Revised	Software Version
N/A	October 29, 2002	BEA WebLogic Server Version 8.1

Contents

About This Document

1. Understanding WebLogic Server Applications

2. Developing WebLogic Server Applications

3. WebLogic Server Application Packaging

4. WebLogic Server Application Classloading

5. Programming Topics

A. 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," describes components of WebLogic Server applications.
- Chapter 2, "Developing WebLogic Server Applications," outlines high-level procedures for creating WebLogic Server applications and helps Java programmers establish their programming environment.
- Chapter 3, "WebLogic Server Application Packaging," provides procedures for packaging WebLogic Server applications.
- Chapter 4, "WebLogic Server Application Classloading," provides an overview of Java classloaders, followed by details about WebLogic Server application classloading.
- Chapter 5, "Programming Topics," covers general WebLogic Server application programming issues, such as logging messages and using threads.
- Appendix A, "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 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.

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 components:

- *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81b/ejb/index.html
- *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs81b/servlet/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81b/jsp/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs81b/webapp/index.html
- *Programming WebLogic JDBC* at http://e-docs.bea.com/wls/docs81b/jdbc/index.html
- *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs81b/webServices/index.html
- *Programming WebLogic J2EE Connectors* at http://e-docs.bea.com/wls/docs81b/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.

Convention	Usage

`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 *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: 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}`
`...`	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

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Understanding WebLogic Server Applications

The following sections provide an overview of WebLogic Server J2EE applications and application components:

What Are WebLogic Server J2EE Applications and Components?

A BEA WebLogic Server™ J2EE application consists of one of the following components running on WebLogic Server:

- Web components—HTML pages, servlets, JavaServer Pages, and related files
- Enterprise Java Beans (EJB) components—entity beans, session beans, and message-driven beans
- Connector component—resource adapters

Components are packaged in Java ARchive (JAR) files—archives created with the Java `jar` utility. JAR files bundle all component files in a directory into a single file, maintaining the directory structure. JAR files also include XML descriptors that instruct WebLogic Server how to deploy the components.

Web applications are packaged in a JAR file with a `.war` extension. Enterprise beans, WebLogic components, and client applications are packaged in JAR files with `.jar` extensions. Resource adapters are packaged in a JAR file with a `.rar` extension.

An enterprise application, consisting of assembled Web application components, EJB components, and resource adapters, is a JAR file with an `.ear` extension. An EAR file contains all of the JAR, WAR, and RAR component archive files for an application and an XML descriptor that describes the bundled components.

To deploy a component, an application, or a resource adapter, you use the Administration Console or the `weblogic.Deployer` command-line utility to upload JAR files to the target WebLogic Server instances.

Client applications that are not Web browsers are Java classes that connect to WebLogic Server using Remote Method Invocation (RMI). A Java client can remotely access Enterprise JavaBeans, JDBC connections, JMS messaging, and other services using access methods such as RMI.

J2EE Platform

WebLogic Server implements Java 2 Platform, Enterprise Edition (J2EE) version 1.3 technologies (`http://java.sun.com/j2ee/sdk_1.3/index.html`). J2EE is the standard platform for developing multitier 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.

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

Note: Because J2EE is backward compatible, you can still run J2EE 1.2 on WebLogic Server 7.0.

Web Application Components

A Web archive (WAR) file has a `.war` extension and contains the components that make up a Web application. A WAR file is deployed as a unit on one or more WebLogic Servers.

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.
- A `weblogic.xml` deployment descriptor, an XML document containing WebLogic Server-specific elements for Web applications.

A Web application might 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. A GenericServlet is protocol independent and can be used in J2EE applications to implement services accessed from other Java classes. An HttpServlet extends GenericServlet with support for the HTTP protocol. 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, called taglibs, using HTML-like tags. The WebLogic JSP compiler, `weblogic.jspc`, translates JSPs into servlets. WebLogic Server automatically compiles JSPs if the servlet class file is not present or is older than the JSP source file.

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.

Web Application Directory Structure

You assemble Web application components in a directory, then package them into a WAR file with the `jar` command.

HTML pages, JSPs, and the non-Java class files they reference are accessed beginning in the top level of the staging directory.

The XML descriptors, compiled Java classes and JSP taglibs are stored in a `WEB-INF` subdirectory at the top level of the staging directory. Java classes include servlets, helper classes and, if desired, precompiled JSPs.

The entire directory, once staged, is bundled into a WAR file using the `jar` command. You can deploy the WAR file alone or packaged in an Enterprise Archive (EAR file) with other application components, including other Web Applications, EJB components, and WebLogic Server components.

See Directory Structure at http://e-docs.bea.com/wls/docs81b/webapp/basics.html#136976 for detailed information on the Web application directory structure.

More Information on Web Application Components

For more information about creating Web application components, see these documents:

- *Programming WebLogic Server HTTP Servlets* at http://e-docs.bea.com/wls/docs81b/servlet/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81b/jsp/index.html
- *Programming JSP Tag Extensions* at http://e-docs.bea.com/wls/docs81b/taglib/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs81b/webapp/index.html

Enterprise JavaBean Components

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

A message-driven bean is an enterprise bean that runs in the EJB container and handles asynchronous messages from a JMS Queue. When a message is received in the JMS Queue, 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. A JMS ServerSessionPool provides a similar capability but does not run in the EJB container.

Enterprise beans are bundled into a JAR file with a `.jar` extension that contains their compiled classes and XML deployment descriptors.

EJB Interfaces

Entity beans and session beans have remote interfaces, home interfaces, and implementation classes provided by the bean developer. (Message-driven beans do not require home or remote interfaces, because they are not accessible outside of the EJB container.)

The remote interface defines the methods a client can call on an entity bean or session bean. The implementation class is the server-side implementation of the remote interface. The home interface provides methods for creating, destroying, and finding enterprise beans. The client accesses instances of an enterprise bean through the bean's home interface.

EJB home and remote interfaces and implementation classes are portable to any EJB container that implements the EJB specification. An EJB developer can supply a JAR file containing just the compiled EJB interfaces and classes and a deployment descriptor.

EJBs and WebLogic Server

J2EE cleanly separates the development and deployment roles to ensure that components are portable between EJB servers that support the EJB specification. Deploying an enterprise bean in WebLogic Server requires running the WebLogic EJB compiler, `weblogic.appc`, to generate classes that enforce the EJB security, transaction, and life cycle policies.

The J2EE-specified deployment descriptor, `ejb-jar.xml`, describes the enterprise beans packaged in an EJB JAR file. 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 for 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 clustering and cache configuration.

For help creating and deploying EJBs, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81b/ejb/index.html.

Connector Component

The WebLogic Server J2EE Connector architecture enables both Enterprise Information Systems (EIS) vendors and third-party application developers to develop resource adapters that can be deployed in any application server supporting the J2EE 1.3 specification from Sun Microsystems. Resource adapters contain the Java, and if necessary, the native components required to interact with the EIS.

A resource adapter deployed in the WebLogic Server environment enables J2EE applications to access a remote EIS system. Developers of WebLogic Server applications can use HTTP servlets, JavaServer Pages (JSPs), Enterprise Java Beans (EJBs), and other APIs to develop integrated applications that use the data and business logic of the EIS.

As is, the basic Resource ARchive (RAR File) or deployment directory cannot be deployed to WebLogic Server. You must first create and configure WebLogic Server-specific deployment properties in the `weblogic-ra.xml` file, and add that file to the deployment directory.

To configure and deploy resource adapters, see *Programming WebLogic J2EE Connectors* at http://e-docs.bea.com/wls/docs81b/jconnector/index.html.

Enterprise Applications

An enterprise J2EE application contains Web and EJB components, deployment descriptors, and archive files. These components are packaged in an Enterprise Archive (EAR) file with an `.ear` extension.

The `META-INF/application.xml` deployment descriptor contains an entry for each Web and EJB component, and additional entries to describe security roles and application resources such as databases.

From the WebLogic Administration Server you use the Administration Console or the `weblogic.Deployer` command line utility to deploy an EAR file on one or more WebLogic Server instances in a domain.

WebLogic Web Services

Web services can be shared by and used as components 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 XML and HTTP, thus making them easily accessible by any user on the Web.

A Web service consists of the following components:

- 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 components (such as Enterprise Java Beans) and packaged as standard J2EE Enterprise Applications.

- A standardized way to transmit 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 way to describe 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.

For information on designing, developing, and invoking WebLogic Web services, see *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs81b/webServices/index.html.

Client Applications

Java clients that access WebLogic Server components 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 use WebLogic Server components indirectly through HTTP requests or RMI requests. The components execute in WebLogic Server, not in the client.

WebLogic Server supports a variety of Java clients, which vary in terms of protocol support and the WebLogic Server classes required on 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*. Small footprint standard and JMS jars—`wlclient.jar` and `wljmsclient.jar` respectively—are provided in the `/server/lib` subdirectory of the WebLogic Server installation directory. Each jar 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 beans running in the business tier, and may, as appropriate

communicate via 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 component, 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*.

The application developer packages client-side applications so they can be deployed on client computers. To simplify maintenance and deployment, it is a good idea to package a client-side application in a JAR file that can be added to the client's classpath along with the appropriate WebLogic jar file.

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

Naming Conventions

WebLogic Server requires you to adhere to the following programmatic naming conventions for WAR, EAR, JAR, and RAR archive files and exploded directories.

- Enterprise JavaBean JAR archived files must end with the `.jar` extension.
- Resource adapter RAR archived files must end with the `.rar` extension.
- Web application WAR archived files must end with the `.war` extension.
- Enterprise application EAR archived files must end with the `.ear` extension.
- Exploded non-archived versions of all of the above archived files must *not* end with the `.jar`, `.rar`, `.war`, or `.ear` extensions respectively.

CHAPTER

2 Developing WebLogic Server Applications

The following sections describe the steps for creating different types of WebLogic Server J2EE applications, setting up a development environment, and preparing to compile Java programs.

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

Establishing a Development Environment

In preparation for developing WebLogic Server applications, you assemble the required software tools and set up an environment for creating, compiling, deploying, testing, and debugging your code.

Software Tools

This section reviews the software required to develop WebLogic Server applications and describes optional tools for development and debugging.

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.

Java Interactive Development Environments (IDEs) such as WebGain VisualCafé usually include a programmer's editor with custom support for Java. An IDE may also have support for creating and deploying servlets and Enterprise JavaBeans on WebLogic Server, which makes it much easier to develop, test, and debug applications.

You can edit HTML or XML pages and JavaServer Pages with a plain text editor, or use a Web page editor such as DreamWeaver.

XML Editor

You use an XML editor to edit the XML files used by WebLogic Server, such as the EJB and Web application deployment descriptors, the `config.xml` file, and so on. WebLogic Server includes the following two XML editors:

- Deployment Descriptor Editor, part of the Administration Console
- BEA XML Editor, a stand-alone Java-based editor

For detailed information about using these XML editors, see "Deployment Tools Reference" in *Deploying WebLogic Server Applications*.

appc Compiler

The appc compiler compiles and generates EJBs and JSPs for deployment. It 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. Finally, the appc compiler 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:

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

appc Ant Task

You can use the following Ant task to invoke the appc compiler:

```
<taskdef name="appc"
classname="weblogic.ant.taskdefs.j2ee.Appc"/>
```

Development WebLogic Server

Never deploy untested code on a WebLogic Server 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.

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.

Even if you do not run a development WebLogic Server 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. Installing WebLogic Server on your development computer makes these files available locally.

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.

JDBC connection pools offer such significant performance advantages that you should only rarely consider writing an application that uses a two-tier JDBC driver directly. On a WebLogic Server cluster, be sure to set up a multipool, which provides load balancing over JDBC connection pools on multiple servers in the cluster.

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, such as WebGain Studio, WebGain StructureBuilder, and BEA WebLogic Integration Kit for VisualAge for Java, to enhance your WebLogic Server development environment.

For more information, 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.

Application Lifecycle Events

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

Four application lifecycle events are provided with WebLogic Server:

- Prestart—the beginning of the prepare phase. You can use the prestart event to establish a connection pool.
- Poststart—the end of the activate phase; the application is deployed.
- Prestop—the beginning of the deactivate phase. You can use the prestop event to disconnect from the database.
- Poststop—the end of the remove phase.

User-defined listeners can be:

- Listeners—attachable to any event. Possible methods for Listeners are:
  - `public void preStart(ApplicationLifecycleEvent evt) {}`
  - `public void postStart(ApplicationLifecycleEvent evt) {}`
  - `public void preStop(ApplicationLifecycleEvent evt) {}`
  - `public void postStop(ApplicationLifecycleEvent evt) {}`
- Startup—attachable to prestart and poststart events.
- Shutdown—attachable to prestop and poststop events.

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

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

Basic Functionality

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

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

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

Listing 2-1 illustrates how you override the `ApplicationLifecycleListener`. In this example, the public class `MyListener` extends `ApplicationLifecycleListener`.

Listing 2-1 MyListener

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyListener extends ApplicationLifecycleListener {
  public void preStart(ApplicationLifecycleEvent evt) {
     System.out.println
     ("MyListener(preStart) -- we should always see you..");
   } // preStart
  public void postStart(ApplicationLifecycleEvent evt) {
```

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

Listing 2-2 illustrates how you implement the Shutdown class. This class is attachable to prestop and poststop events. In this example, the public class `MyShutdown` extends `ApplicationLifecycleListener`.

Listing 2-2 MyShutdown

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyShutdown extends ApplicationLifecycleListener {
   public static void main(String[] args) {
     System.out.println
```

```
      ("MyShutdown(main): in main .. should be for post-stop");
   } // main
}
```

Listing 2-3 illustrates how you implement the Startup class. This class is attachable to prestart and poststart events.. In this example, the public class `MyStartup` extends `ApplicationLifecycleListener`.

Listing 2-3 MyStartup

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

Configuring Lifecycle Events: URI Parameter

The following are examples illustrating how you configure the application lifecycle events in the `application.xml` deployment descriptor file. The URI parameter is not required. You can place classes anywhere in the application `$CLASSPATH`. However, you must ensure that the class locations are defined in the `$CLASSPATH`. You can place listeners in `APP-INF/classes` or `APP-INF/lib`, if these directories are present in the EAR. In this case, they are automatically included in the `$CLASSPATH`.

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

Listing 2-4 Configuring Application Lifecycle Events without URI Parameter

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

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

Listing 2-5 Configuring Application Lifecycle Events without URI Parameter

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

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

Creating Web Applications: Main Steps

Here are the main steps for creating a Web application:

1. Create the HTML pages and JavaServer Pages (JSPs) that make up the Web interface of the Web application. Typically, Web designers create these parts of a Web application.

   For detailed information about creating JSPs, refer to *Programming WebLogic JSP*.

2. Write the Java code for the servlets and the JSP taglibs referenced in JSPs. Typically, Java programmers create these parts of a Web application.

   For detailed information about creating servlets, refer to *Programming WebLogic HTTP Servlets*.

3. Compile the servlets into class files.

   For detailed information about compiling, refer to "Compiling Java Code" on page 2-22.

4. Arrange the resources (servlets, JSPs, static files, and deployment descriptors) in the prescribed directory format. For more information on the Web application directory structure, see "Web Application Basics" in *Developing Web Applications for WebLogic Server*.

5. Create the `web.xml` and `weblogic.xml` deployment descriptors.

   The `web.xml` file defines each servlet and JSP page and enumerates enterprise beans referenced in the Web application. The `weblogic.xml` file adds additional deployment information for WebLogic Server.

Create the `web.xml` and `weblogic.xml` deployment descriptors manually or using WebLogic Builder. For detailed information, refer to *WebLogic Builder Online Help*. See *Developing Web Applications for WebLogic Server* for detailed information on the elements in these deployment descriptors.

6. Package the HTML pages, servlet class files, JSP files, `web.xml` file, and `weblogic.xml` file into a WAR file.

   Create a Web application staging directory and save the JSPs, HTML pages, and multimedia files referenced by the pages in the top level of the staging directory.

   Store compiled servlet classes, taglibs, and, if desired, servlets compiled from JSP pages are stored under a `WEB-INF` directory in the staging directory. When the Web application components are all in place in the staging directory, you create the WAR file with the JAR command.

   For detailed information on packaging, refer to "WebLogic Server Application Packaging" on page 3-1.

7. Auto-deploy the WAR file on WebLogic Server for testing purposes.

   For detailed information about auto-deploying components and applications, refer to "Deployment Tool Reference" in *Deploying WebLogic Server Applications*.

   While you are testing the Web application, you might need to edit the Web application deployment descriptors. You can do this manually or use WebLogic Builder.

   For detailed information, refer to *WebLogic Builder Online Help*. See *Developing Web Applications for WebLogic Server* for detailed information on the elements in these deployment descriptors.

8. Deploy the WAR file on the WebLogic Server for production use or include it in an Enterprise ARchive (EAR) file to be deployed as part of an enterprise application.

   Refer to *Deploying WebLogic Server Applications* for detailed information about deploying components and applications.

Creating Enterprise JavaBeans: Main Steps

Creating an Enterprise JavaBean requires creating the classes for the particular EJB (session, entity, or message-driven) and the EJB-specific deployment descriptors, and then packaging everything into an EAR file to be deployed on WebLogic Server.

Here are the main steps for creating an Enterprise JavaBean:

1. Write the Java code for the various classes required by each type of EJB (session, entity, or message-driven) in accordance with the EJB specification. For example, session and entity EJBs require the following three classes:
   - An EJB home interface
   - A remote interface for the EJB
   - An implementation class for the EJB

   Message-driven beans, however, require only an implementation class.
2. Compile the Java code using a standard compiler for the interfaces and implementation into class files.

   For instructions on compiling, refer to "Compiling Java Code" on page 2-22.
3. Create the EJB-specific deployment descriptors:
   - `ejb-jar.xml` describes the EJB type and its deployment properties using a standard DTD from Sun Microsystems.
   - `weblogic-ejb-jar.xml` adds additional WebLogic Server-specific deployment information.
   - `weblogic-cmp-rdbms-jar.xml` maps a container-managed entity EJB to tables in a database. This file can must have a different name for each container-managed persistence (CMP) bean packaged in a JAR file. The name of the file is specified in the bean's entry in the `weblogic-ejb.jar` file.

   Component deployment descriptors are XML documents that provide information needed to deploy the application in WebLogic Server. The J2EE specifications define the contents of some deployment descriptors, such as `ejb-jar.xml` and `web.xml`. Additional deployment descriptors supplement the

J2EE-specified descriptors with information required to deploy components in WebLogic Server.

Create and edit the XML deployment descriptors manually, or use WebLogic Builder to automatically generate them. For more information, refer to *Deploying WebLogic Server Applications*.

For detailed information about the elements in the EJB-specific deployment descriptors and how to create the files by hand, refer to *Programming WebLogic Enterprise JavaBeans*.

4. Package the class files and deployment descriptors into a JAR file.

   Create an EJB staging directory. Place the compiled Java classes in the staging directory and the deployment descriptors in a subdirectory called `META-INF`. Then run the `weblogic.ejbc` EJB compiler to generate classes that enforce the EJB security, transaction, and lifecycle policies. Then you create the EJB archive by executing a `jar` command like the following in the staging directory:

   ```
   jar cvf myEJB.jar *
   ```

   For detailed information about creating the EJB JAR file, refer to "WebLogic Server Application Packaging" on page 3-1.

5. Auto-deploy the EJB JAR file on WebLogic Server for testing purposes.

   For detailed information about auto-deploying components and applications, refer to "Deployment Tool Reference" in *Deploying WebLogic Server Applications*.

   While you are testing the EJB, you might need to edit the EJB deployment descriptors. You can do this manually or use WebLogic Builder.

   For detailed information, refer to *WebLogic Builder Online Help*. See *Developing Web Applications for WebLogic Server* for detailed information on the elements in these deployment descriptors.

6. Deploy the JAR file on WebLogic Server for production use or include it in an Enterprise ARchive (EAR) file to be deployed as part of an enterprise application.

   Refer to *Deploying WebLogic Server Applications* for detailed information about deploying components and applications.

Creating Resource Adapters: Main Steps

Creating a resource adapter requires creating the classes for a resource adapter and the connector-specific deployment descriptors, and then packaging everything into a resource adapter archive (RAR) file to be deployed on WebLogic Server.

Creating a New Resource Adapter (RAR)

The following are the main steps for creating a resource adapter (RAR):

1. Write the Java code for the various classes required by resource adapter (ConnectionFactory, Connection, and so on) in accordance with the J2EE Connector Specification, Version 1.0, Final Release (http://java.sun.com/j2ee/download.html#connectorspec).

   When implementing a resource adapter, you must specify classes in the `ra.xml` file. For example:

   - `<managedconnectionfactory-class>com.sun.connector.blackbox.LocalTxManagedConnectionFactory</managedconnectionfactory-class>`
   - `<connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>`
   - `<connectionfactory-impl-class>com.sun.connector.blackbox.JdbcDataSource</connectionfactory-impl-class>`
   - `<connection-interface>java.sql.Connection</connection-interface>`
   - `<connection-impl-class>com.sun.connector.blackbox.JdbcConnection</connection-impl-class>`

2. Compile the Java code using a standard compiler for the interfaces and implementation into class files.

   For instructions on compiling, refer to "Compiling Java Code" on page 2-22.

3. Create the resource connector-specific deployment descriptors:

   - `ra.xml` describes the resource adapter-related attributes type and its deployment properties using a standard DTD from Sun Microsystems.

- `weblogic-ra.xml` adds additional WebLogic Server-specific deployment information.

For detailed information about creating connector-specific deployment descriptors, refer to *Programming WebLogic Server J2EE Connectors*.

4. Package the Java classes into a Java archive (JAR) file.

   The first step in creating a JAR file is to create a connector staging directory anywhere on your hard drive. Place the JAR file in the staging directory and the deployment descriptors in a subdirectory called `META-INF`.

   Then you create the resource adapter archive by executing a `jar` command similar to the following in the staging directory:

   ```
   jar cvf myRAR.rar *
   ```

   For detailed information about creating the resource adapter RAR archive file, refer to "WebLogic Server Application Packaging" on page 3-1.

5. Auto-deploy the RAR resource adapter archive file on WebLogic Server for testing purposes.

   For detailed information about auto-deploying components and applications, refer to "Tools for Deploying" in *Deploying WebLogic Server Applications*.

   While you are testing the resource adapter, you might need to edit the resource adapter deployment descriptors. You can do this manually or use WebLogic Builder.

   For detailed information, refer to *WebLogic Builder Online Help*. See *Programming WebLogic Server J2EE Connectors* for detailed information on the elements in these deployment descriptors.

6. Deploy the RAR resource adapter archive file on WebLogic Server or include it in an enterprise archive (EAR) file to be deployed as part of an enterprise application.

   Refer to *Deploying WebLogic Server Applications* for detailed information about deploying components and applications.

Modifying an Existing Resource Adapter (RAR)

The following is an example of how to take an existing resource adapter (RAR) and modify it for deployment to WebLogic Server. This involves adding the `weblogic-ra.xml` deployment descriptor and repacking.

1. Create a temporary directory anywhere on your hard drive to stage the resource adapter:

   ```
   mkdir c:/stagedir
   ```

2. Copy the resource adapter that you will deploy into the temporary directory:

   ```
   cp blackbox-notx.rar c:/stagedir
   ```

3. Extract the contents of the resource adapter archive:

   ```
   cd c:/stagedir
   jar xf blackbox-notx.rar
   ```

   The staging directory should now contain the following:

   - A `jar` file containing Java classes that implement the resource adapter
   - A `META-INF` directory containing the files: `Manifest.mf` and `ra.xml`

   Execute these commands to see these files:

   ```
   c:/stagedir> ls
          blackbox-notx.rar
          META-INF
   c:/stagedir> ls META-INF
          Manifest.mf
          ra.xml
   ```

4. Create the `weblogic-ra.xml` file. This file is the WebLogic-specific deployment descriptor for resource adapters. In this file, you specify parameters for connection factories, connection pools, and security mappings.

   Refer to *Programming WebLogic Server J2EE Connectors* for more information on the `weblogic-ra.xml` DTD.

5. Copy the `weblogic-ra.xml` file into the temporary directory's `META-INF` subdirectory. The `META-INF` directory is located in the temporary directory where you extracted the RAR file or in the directory containing a resource adapter in exploded directory format. Use the following command:

```
cp weblogic-ra.xml c:/stagedir/META-INF

c:/stagedir> ls META-INF

       Manifest.mf

       ra.xml

       weblogic-ra.xml
```

6. Create the resource adapter archive:

```
jar cvf blackbox-notx.rar -C c:/stagedir
```

7. Deploy the resource adapter to WebLogic Server.

   For detailed information about deploying components and applications, refer to "Tools for Deploying" in *Deploying WebLogic Server Applications*.

Creating WebLogic Server Enterprise Applications: Main Steps

Creating a WebLogic Server enterprise application requires creating Web, EJB, and Connector (Resource Adapter) components, deployment descriptors, and archive files. The result is an enterprise application archive (EAR file) that can be deployed on WebLogic Server.

Here are the main steps for creating a WebLogic Server enterprise application:

1. Create Web, EJB, and Connector components for your application.

   Programmers create servlets, EJBs, and Connectors using the J2EE APIs for these components. Web designers create Web pages using HTML/XML and JavaServer Pages.

For overview information about creating Web, EJB, and Connector components, respectively refer to "Creating Web Applications: Main Steps" on page 2-12, "Creating Enterprise JavaBeans: Main Steps" on page 2-14, and "Creating Resource Adapters: Main Steps" on page 2-16.

For detailed information about creating the Java code that makes up the Web, EJB, and Connector components, refer to *Programming WebLogic Enterprise JavaBeans*, *Programming WebLogic HTTP Servlets*, *Programming WebLogic JSP*, and *Programming WebLogic Server J2EE Connectors*.

2. Create Web, EJB, and Connector deployment descriptors.

   Component deployment descriptors are XML documents that provide information needed to deploy the application in WebLogic Server. The J2EE specifications define the contents of some deployment descriptors, such as `ejb-jar.xml`, `web.xml`, and `ra.xml`. Additional deployment descriptors supplement the J2EE-specified descriptors with information required to deploy components in WebLogic Server.

   Create and edit the XML deployment descriptors manually, or use WebLogic Builder to automatically generate them. For more information, refer to *Deploying WebLogic Server Applications*.

   For detailed information about the various deployment descriptor elements, refer to *Developing Web Applications for WebLogic Server*, *Programming WebLogic Enterprise JavaBeans*, and *Programming WebLogic Server J2EE Connectors*.

3. Package the Web, EJB, and Connector components into their component archive files.

   Component archives are JAR files containing all component files, including deployment descriptors. You package Web components into a WAR file, EJB components into an EJB JAR file, and Connector components into a RAR file.

   Refer to "WebLogic Server Application Packaging" on page 3-1 for detailed information for creating component archives.

4. Create the enterprise application deployment descriptor.

   The enterprise application deployment descriptor, `application.xml`, lists individual components that are assembled together in an application.

   Create the `application.xml` deployment descriptor manually, or use WebLogic Builder to automatically generate it. For more information, refer to *Deploying WebLogic Server Applications*.

Refer to "application.xml Deployment Descriptor Elements" on page A-1 for detailed information about the elements of the `application.xml` file.

5. Package the enterprise application into an EAR file.

   Package the Web, EJB, and Connector component archives along with the enterprise application deployment descriptor into an enterprise archive (`.ear` extension) file. This is the file that is deployed on WebLogic Server. WebLogic Server uses the `application.xml` deployment descriptor to locate and deploy the individual components packaged in the EAR file.

   For detailed information about creating the EAR file, see "WebLogic Server Application Packaging" on page 3-1.

6. For testing purposes, auto-deploy the EAR enterprise application on WebLogic Server.

   While you are testing the enterprise application, you might need to edit the enterprise application deployment descriptor. You can do this manually or use WebLogic Builder.

   For detailed information on WebLogic Builder, refer to *WebLogic Builder Online Help*.

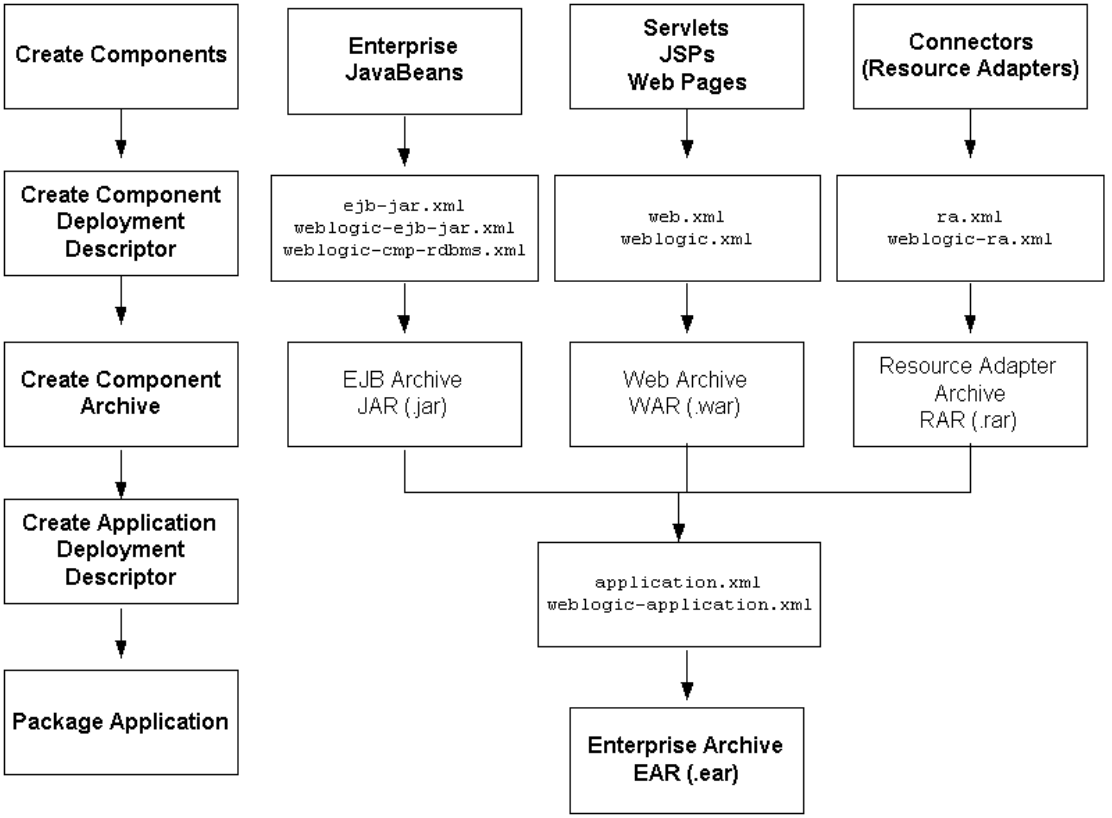
   Refer to "application.xml Deployment Descriptor Elements" on page A-1 for detailed information about the elements of the `application.xml` deployment descriptor file.

7. For production purposes, deploy the EAR file on WebLogic Server.

   For detailed information about deploying components and applications, refer to "Deployment Tools Reference" in *Deploying WebLogic Server Applications*.

Figure 2-1 illustrates the process for developing and packaging WebLogic Server enterprise applications.

Figure 2-1 Creating Enterprise Applications



Compiling Java Code

Compiling Java code for WebLogic Server is the same as compiling any other Java code. To compile successfully, you must:

- Place a standard Java compiler in your search path.
- Set your classpath so that the Java compiler can find all of the dependent classes.
- Specify the output directories for the compiled classes.

- Set your environment by creating a command file or script to set variables in your environment, which you can pass to the compiler.

Creating Compile Scripts Using Apache Ant

The preferred BEA method for compiling is using Apache Ant. Apache Ant is a Java-based build tool. One of the benefits of using Ant is that is it extended using Java classes, rather than shell-based commands. Another benefit is that Ant is a cross-platform tool.

Developers write Ant build scripts using eXtensible Markup Language (XML). XML tags define the targets to build, dependencies among targets, and tasks to execute in order to build the targets.

Instead of a model where it is extended with shell-based commands, Ant is extended using Java classes. Ant libraries are bundled with WebLogic Server to make it easier for our customers to build Java applications out of the box.

In order 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\server\config\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\server\src\examples\ejb20\basic\beanManaged\build.xml`

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

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

Putting the Java Tools in Your Search Path

Make sure the operating system can find the compiler and other JDK tools by adding it to the `%PATH%` environment variable in your command shell. If you are using the JDK, the tools are in the `bin` subdirectory of the JDK directory. To use an alternative compiler, such as the `sj` compiler from WebGain VisualCafé, add the directory containing that compiler to your search path.

For example, if the JDK is installed in `\usr\local\java\java141` on your UNIX file system, use a command such as the following to add `javac` to your path in a Bourne shell or shell script:

```
PATH=\usr\local\java\java141\bin:$PATH; export $PATH
```

To add the WebGain `sj` compiler to your path on Windows NT, Windows 2000 or Windows 2000 XP, use a command such as the following in a command shell or in a command file:

```
PATH=c:\VisualCafe\bin;%PATH%
```

If you are using an IDE, see the IDE documentation for help setting up an equivalent search path.

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\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.

  Include in your classpath the target directories where the compiler writes the classes you are compiling so that the compiler can locate all of the interdependent classes in your application. The next section has more information on target directories.

Setting Target Directories for Compiled Classes

The Java compiler writes class files in the same directory with the Java source unless you specify an output directory for the compiled classes. If you specify the output directory, the compiler stores the class file in a directory structure that matches the package name. This allows you to compile Java classes into the correct locations in the staging directory you use to package your application. If you do not specify an output directory, you have to move files around before you can create the JAR file that contains your packaged component.

J2EE applications consist of modules assembled into an application and deployed on one or more WebLogic Servers or WebLogic Server clusters. Each module should have its own staging directory so that it can be compiled, packaged, and deployed independently from other modules. For example, you can package EJBs in a separate module, Web components in a separate module, and other server-side classes in another module.

See the `setExamplesEnv` scripts in the `samples\server\config\examples` directory of the WebLogic Server distribution for an example of setting up target directories for the compiler. The scripts set the following variables:

`CLIENT_CLASSES`
: `samples\server\stage\examples\clientclasses`
Directory where compiled client classes are written for the Examples domain. These classes are usually standalone Java programs that connect to WebLogic Server.

`SERVER_CLASSES`
: `samples\server\stage\examples\serverclasses` by default.
Directory where server-side classes are written for the Examples domain. Include startup classes and other Java classes that must be in the WebLogic Server CLASSPATH when the server starts up. Application classes usually should not be compiled into this directory, because the classes in this directory cannot be redeployed without restarting WebLogic Server.

`EX_WEBAPP_CLASSES`
: `samples\server\stage\examples\applications\examplesWebApp\WEB-INF\classes`
Directory where classes used by a Web Application are written for the Examples domain.

`APPLICATIONS`
: `samples\server\config\examples\applications`
Applications directory for the Examples domain. This variable is not used to specify a target for the Java compiler. It is used as a convenient reference to the applications directory in copy commands that move files from source directories into the applications directory. For example, if you have HTML, JSP, and image files in your source tree, you can use the variable in a copy command to install them in your development server.

These environment variables are passed to the compiler in commands such as the following command for Windows:

```
javac -d %SERVER_CLASSES% *.java
```

If you do not use an IDE, consider writing an Apache Ant script to compile and package your components and applications.

Auto-Deployment for Development Enviroments

Auto-deployment is a method for quickly deploying an application on the administration server. It is recommended that this method be used only in a single-server development environment for testing an application. Use of auto-deployment in a production environment or for deployment of components on managed servers is not recommended.

If auto-deployment is enabled, when an application is copied into the `\applications` directory of the administration server, the administration server detects the presence of the new application and deploys it automatically (if the administration server is running). If WebLogic Server is not running when you copy the application to the `\applications` directory, the application is deployed the next time the WebLogic Server is started. Auto-deployment deploys only to the administration server

Note: Due to the strict file locking limitations of Windows NT, if your applications are exploded, all the components within your applications must also be exploded. In other words, WebLogic Server cannot support a JAR file within an exploded application or component.

Enabling and Disabling Auto-Deployment

You can run WebLogic Server in two different modes: development and production. You use development mode to test your applications. Once they are ready for a production environment, you deploy your applications on a server that is started in production mode.

Development mode enables a WebLogic Server to automatically deploy and update applications that are in the domain_name/applications directory (where domain_name is the name of a WebLogic Server domain). In other words, development mode lets you use auto-deploy.

Production mode disables the auto-deployment feature. Instead, you must use the WebLogic Server Administration Console or the `weblogic.Deployer` tool.

By default, a WebLogic Server runs in development mode. To specify the mode for a server, do one of the following:

If you use the startWebLogic startup script, edit the script and set the STARTMODE variable as follows:

`STARTMODE` = false enables deployment mode

`STARTMODE` = true enables production mode

If you start a server entering the `weblogic.Server` command directly on the command line, use the -Dweblogic.ProductionModeEnabled option as follows:

```
-Dweblogic.ProductionModeEnabled=false enables deployment mode
```

```
-Dweblogic.ProductionModeEnabled=true enables production mode
```

Auto-Deploying Applications

This is a convenience feature for deploying applications during development. It allows deploying of applications or individual J2EE modules to the administration server just by copying the deployment into a predefined auto-deployment directory. This directory is located under the domain directory, e.g., `mydomain/applications`.

Stopping and Redeploying Archived Applications

An application or its component that was auto-deployed can be dynamically redeployed while the server is running. To dynamically redeploy a JAR, WAR or EAR file, simply copy the new version of the file over the existing file in the `\applications` directory.

This feature is useful for developers who can simply add the copy to the `\applications` directory as the last step in their makefile, and the server will then be updated.

If you delete the application from the `\applications` directory, the application will be stopped and removed from the configuration.

Redeploying Applications in Exploded Format

You can also dynamically redeploy applications or components that have been auto-deployed in exploded format. When an application has been deployed in exploded format, the administration server periodically looks for a file named `REDEPLOY` in the exploded application directory. If the timestamp on this file changes, the administration server redeploys the exploded directory.

If you want to update files in an exploded application directory, do the following:

1. When you first deploy the exploded application, create an empty file named `REDEPLOY`, and place it in the `WEB-INF` or `META-INF` directory, depending on the application type you are deploying:

   An exploded application contains a `META-INF` top-level directory; this contains the `application.xml` file.

An exploded Web application contains a `WEB-INF` top-level directory; this contains the `web.xml` file.

An exploded EJB application contains a `META-INF` F top-level directory; this contains the `ejb-jar.xml` file.

An exploded connector contains a `META-INF` top-level directory; this contains the `ra.xml` file.

2. To update the exploded application, copy the updated files over the existing files in that directory.
3. After copying the new files, modify the `REDEPLOY` file in the exploded directory to alter its timestamp.

When the administration server detects the changed timestamp, it redeploys the contents of the exploded directory.

CHAPTER

3 WebLogic Server Application Packaging

The following sections describe how to package WebLogic Server components. You must package components before you deploy them to WebLogic Server.

Packaging Overview

WebLogic Server J2EE applications are packaged according to J2EE specifications. J2EE defines component behaviors and packaging in a generic, portable way, postponing run-time configuration until the component is actually deployed on an application server.

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

For each component type, the specifications define the files required and their location in the directory structure. Components and applications may include Java classes for EJBs and servlets, resource adapters, Web pages and supporting files, XML-formatted deployment descriptors, and JAR files containing other components.

An application that is ready to deploy on WebLogic Server may require WebLogic-specific deployment descriptors and, possibly, *container* classes generated with the WebLogic EJB, RMI, or JSP compilers.

For more information, refer to the J2EE 1.3 specification at:
http://java.sun.com/j2ee/download.html#platformspec

JAR Files

A file created with the Java `jar` tool bundles the files in a directory into a single Java ARchive (JAR) file, maintaining the directory structure. 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 components on WebLogic Server in either an "exploded" directory or a JAR file.

JAR files are convenient for packaging components and applications for distribution. They are easier to copy, they use up fewer file handles than an exploded directory, and they can save disk space with file compression. If your Administration Server manages

a domain with multiple WebLogic Servers, you can only deploy JAR or EAR files, because the Administration Console does not copy expanded directories to Managed Servers.

The `jar` utility is in the `bin` directory of your Java Development Kit. If you have `javac` in your path, you also have `jar` in your path. The `jar` command syntax and behavior is similar to the UNIX `tar` command.

The most common usages of the `jar` command are:

`jar cf` *`jar-file files ...`*
: Creates a JAR file named *`jar-file`* containing listed files. If you include a directory in the list of files, all files in that directory and its subdirectories are added to the JAR file.

`jar xf` *`jar-file`*
: Extract (unbundle) a JAR file in the current directory.

`jar tf` *`jar-file`*
: List (tell) the contents of a JAR file.

The first flag specifies the operation: **c**reate, e**x**tract, or list (tell). The `f` flag must be followed by a JAR file name. Without the `f` flag, `jar` reads or writes JAR file contents on *`stdin`* or *`stdout`* which is usually not what you want. See the documentation for the JDK utilities for more about `jar` command options.

XML Deployment Descriptors

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

Table 3-1 lists the types of components and applications and their J2EE-standard and WebLogic-specific deployment descriptors.

Table 3-1 J2EE and WebLogic Deployment Descriptors

Component or Application	Scope	Deployment Descriptors
Web Application	J2EE	`web.xml`
	WebLogic	`weblogic.xml`
Enterprise Bean	J2EE	`ejb-jar.xml`
	WebLogic	`weblogic-ejb-jar.xml` `weblogic-cmp-rdbms-jar.xml`
Resource Adapter	J2EE	`ra.xml`
	WebLogic	`weblogic-ra.xml`
Enterprise Application	J2EE	`application.xml`
	WebLogic	`weblogic-application.xml`
Client Application	J2EE	`application-client.xml`
	WebLogic	`client-application.runtime.xml`

When you package a component 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 create the deployment descriptors manually, or you can use WebLogic-specific Java-based utilities to automatically generate them for you. For more information about generating deployment descriptors, see "Automatically Generating Deployment Descriptors" on page 3-5.

If you receive a J2EE-compliant JAR file from a developer, it already contains J2EE-defined deployment descriptors. To deploy the JAR file on WebLogic Server, you extract the contents of the JAR file into a directory, add the WebLogic-specific deployment descriptors and any generated container classes, and then create a new JAR file containing the old and new files. Note that the JAR utility contains a "u" option, which allows you to change or add files directly to an existing JAR.

Automatically Generating Deployment Descriptors

WebLogic Server includes a set of Java-based utilities that automatically generate the deployment descriptors for the following J2EE components: Web applications, Enterprise JavaBeans (version 2.0).

These utilities examine the objects you have assembled in a staging directory and build the appropriate deployment descriptors based on the servlet classes, EJB classes, and so on. The utilities generate both the standard J2EE and WebLogic-specific deployment descriptors for each component.

WebLogic Server includes the following utilities:

- `weblogic.marathon.ddinit.WebInit`

  Creates the deployment descriptors for Web Applications.

- `weblogic.marathon.ddinit.EJBInit`

  Creates the deployment descriptors for Enterprise JavaBeans 2.0.

Note: Although `DDInit` attempts to create deployment descriptor files that are complete and accurate for your component or application, the utilities must guess at the value of many of the required elements. Often this guess is wrong, causing WebLogic Server to return an error when you deploy the component or application. In this case, you must undeploy the component or application, edit the deployment descriptor using the Deployment Descriptor Editor of the Administration Console, and then redeploy it. For details on using the Deployment Descriptor Editor, see "Editing Deployment Descriptors."

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 two tools for editing the deployment descriptors of WebLogic Server applications and components:

- BEA XML Editor
- Deployment Descriptor Editor from within the Administration Console

Use either editor to update existing elements in, add new elements to, and delete existing elements from the following deployment descriptors:

- `web.xml`
- `weblogic.xml`
- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml`
- `ra.xml`
- `weblogic-ra.xml`
- `application.xml`
- `weblogic-application.xml`
- `application-client.xml`
- `client-application.runtime.xml`

Using the BEA XML Editor

To edit XML files, use the BEA XML Editor, an entirely Java-based XML stand-alone editor. It is a simple, user-friendly tool for creating and editing XML files. It displays XML file contents both as a hierarchical XML tree structure and as raw XML code. This dual presentation of the document gives you a choice of editing:

- The hierarchical tree view allows structured, constrained editing, with a set of allowable functions at each point in the hierarchical XML tree structure. The allowable functions are syntactically dictated and in accordance with the XML document's DTD or schema, if one is specified.
- The raw XML code view allows free-form editing of the data.

BEA XML Editor can validate XML code according to a specified DTD or XML schema.

For more documentation about using the BEA XML Editor and to download it, visit *BEA dev2dev Online* at `http://developer.bea.com/tools/utilities.jsp`.

About 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 one the number of EJB files you need to edit and maintain.

For more information about EJBGen, see "EJBGen" in *Programming WebLogic Enterprise JavaBeans.*

Using the Administration Console Deployment Descriptor Editor

The Administration Console Deployment Descriptor Editor looks very much like the main Administration Console: the left pane lists the elements of the deployment descriptor files in tree form and the right pane contains the form for updating a particular element.

When you use the editor, you can either update the in-memory deployment descriptor only, or update both the in-memory and disk files. When you click the Apply button after updating a particular element, or the Create button to create a new element, only

the deployment descriptor in WebLogic Server's memory is updated; the change has not yet been written to disk. To do this, click the Persist button. If you do not explicitly persist the changes to disk, the changes are lost when you stop and restart WebLogic Server.

Editing EJB Deployment Descriptors

This section describes the procedure for editing the following EJB deployment descriptors using the Administration Console Deployment Descriptor Editor:

- `ejb-jar.xml`
- `weblogic-ejb-jar.xml`
- `weblogic-cmp-rdbms-jar.xml`

For detailed information about the elements in the EJB-specific deployment descriptors, refer to *Programming WebLogic Enterprise JavaBeans*.

To edit the EJB deployment descriptors:

1. Invoke the Administration Console in your browser using the following URL:

   `http://`*`host`*`:`*`port`*`/console`

   where *`host`* refers to the name of the computer upon which WebLogic Server is running and *`port`* refers to the port number to which it is listening.

2. Click to expand the Deployments node in the left pane.
3. Click to expand the EJB node under the Deployments node.
4. Right-click the name of the EJB whose deployment descriptors you want to edit and choose Edit EJB Descriptor from the drop-down menu. The Administration Console window appears in a new browser.

   The left pane contains a tree structure that lists all the elements in the three EJB deployment descriptors and the right pane contains a form for the descriptive elements of the `ejb-jar.xml` file.

5. To edit, delete, or add elements in the EJB deployment descriptors, click to expand the node in the left pane that corresponds to the deployment descriptor file you want to edit, as described in the following list:

   - The EJB JAR node contains the elements of the `ejb-jar.xml` deployment descriptor.

- The WebLogic EJB Jar node contains the elements of the `weblogic-ejb-jar.xml` deployment descriptor.
- The container-managed persistence (CMP) node contains the elements of the `weblogic-cmp-rdbms-jar.xml` deployment descriptor.

6. To edit an existing element in one of the EJB deployment descriptors, follow these steps:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the element you want to edit.
   b. Click the element. A form appears in the right pane that lists either its attributes or sub-elements.
   c. Edit the text in the form in the right pane.
   d. Click Apply.
7. To add a new element to one of the EJB deployment descriptors, follow these steps:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to create.
   b. Right-click the element and chose Configure a New *Element* from the drop-down menu.
   c. Enter the element information in the form that appears in the right pane.
   d. Click Create.
8. To delete an existing element from one of the EJB deployment descriptors, follow these steps:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to delete.
   b. Right-click the element and chose Delete *Element* from the drop-down menu.
   c. Click Yes to confirm that you want to delete the element.
9. Once you make all your changes to the EJB deployment descriptors, click the root element of the tree in the left pane. The root element is the either the name of the EJB JAR archive file or the display name of the EJB.

10. Click Validate if you want to ensure that the entries in the EJB deployment descriptors are valid.
11. Click Persist to write your edits of the deployment descriptor files to disk in addition to WebLogic Server memory.

Editing Web Application Deployment Descriptors

This section describes the procedure for editing the `web.xml` and `weblogic.xml` Web application deployment descriptors using the Administration Console Deployment Descriptor Editor.

See *Developing Web Applications for WebLogic Server* for detailed information on the elements in the Web application deployment descriptors.

To edit the Web application deployment descriptors:

1. Invoke the Administration Console in your browser:

   `http://`*`host`*`:`*`port`*`/console`

   where *`host`* refers to the name of the computer upon which WebLogic Server is running and *`port`* refers to the port number to which it is listening.
2. Click to expand the Deployments node in the left pane.
3. Click to expand the Web Applications node under the Deployments node.
4. Right-click the name of the Web application whose deployment descriptors you want to edit and choose Edit Web Application Descriptor from the drop-down menu. The Administration Console window appears in a new browser.

   The left pane contains a tree structure that lists all the elements in the two Web application deployment descriptors and the right pane contains a form for the descriptive elements of the `web.xml` file.
5. To edit, delete, or add elements in the Web application deployment descriptors, click to expand the node in the left pane that corresponds to the deployment descriptor file you want to edit:
   - The Web App Descriptor node contains the elements of the `web.xml` deployment descriptor.
   - The WebApp Ext node contains the elements of the `weblogic.xml` deployment descriptor.

6. To edit an existing element in one of the Web application deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the element you want to edit.
   b. Click the element. A form appears in the right pane that lists either its attributes or sub-elements.
   c. Edit the text in the form in the right pane.
   d. Click Apply.
7. To add a new element to one of the Web application deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to create.
   b. Right-click the element and chose Configure a New *Element* from the drop-down menu.
   c. Enter the element information in the form that appears in the right pane.
   d. Click Create.
8. To delete an existing element from one of the Web application deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to delete.
   b. Right-click the element and choose Delete *Element* from the drop-down menu.
   c. Click Yes to confirm that you want to delete the element.
9. Once you make all your changes to the Web application deployment descriptors, click the root element of the tree in the left pane. The root element is the either the name of the Web application WAR archive file or the display name of the Web application.
10. Click Validate to ensure that the entries in the Web application deployment descriptors are valid.
11. Click Persist to write your edits of the deployment descriptor files to disk in addition to WebLogic Server memory.

Editing Resource Adapter Deployment Descriptors

This section describes the procedure for editing the `ra.xml` and `weblogic-ra.xml` resource adapter deployment descriptors using the Administration Console Deployment Descriptor Editor.

For detailed information about the elements in the resource adapter deployment descriptors, refer to *Programming WebLogic J2EE Connectors*.

To edit the resource adapter deployment descriptors:

1. Invoke the Administration Console in your browser:

   `http://`*`host`*`:`*`port`*`/console`

   where *`host`* refers to the name of the computer upon which WebLogic Server is running and *`port`* refers to the port number to which it is listening.

2. Click to expand the Deployments node in the left pane.
3. Click to expand the Connectors node under the Deployments node.
4. Right-click the name of the resource adapter whose deployment descriptors you want to edit and choose Edit Connector Descriptor from the drop-down menu. The Administration Console window appears in a new browser.

   The left pane contains a tree structure that lists all the elements in the two resource adapter deployment descriptors and the right pane contains a form for the descriptive elements of the `ra.xml` file.

5. To edit, delete, or add elements in the resource adapter deployment descriptors, click to expand the node in the left pane that corresponds to the deployment descriptor file you want to edit:
   - The RA node contains the elements of the `ra.xml` deployment descriptor.
   - The WebLogic RA node contains the elements of the `weblogic-ra.xml` deployment descriptor.
6. To edit an existing element in one of the resource adapter deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the element you want to edit.
   b. Click the element. A form appears in the right pane that lists either its attributes or sub-elements.

c. Edit the text in the form in the right pane.

d. Click Apply.

7. To add a new element to one of the resource adapter deployment descriptors:

   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to create.

   b. Right-click the element and chose Configure a New *Element* from the drop-down menu.

   c. Enter the element information in the form that appears in the right pane.

   d. Click Create.

8. To delete an existing element from one of the resource adapter deployment descriptors:

   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to delete.

   b. Right-click the element and chose Delete *Element* from the drop-down menu.

   c. Click Yes to confirm that you want to delete the element.

9. Once you make all your changes to the resource adapter deployment descriptors, click the root element of the tree in the left pane. The root element is the either the name of the resource adapter RAR archive file or the display name of the resource adapter.

10. Click Validate to ensure that the entries in the resource adapter deployment descriptors are valid.

11. Click Persist to write your edits of the deployment descriptor files to disk in addition to WebLogic Server memory.

Editing Enterprise Application Deployment Descriptors

This section describes the procedure for editing the Enterprise Application deployment descriptors (`application.xml` and `weblogic-application.xml`) using the Administration Console Deployment Descriptor Editor.

Refer to "application.xml Deployment Descriptor Elements" in Appendix A, "Application Deployment Descriptor Elements," for detailed information about the `application.xml` and `weblogic-application.xml` files.

Note: The following procedure describes only how to edit the `application.xml` and `weblogic-application.xml` files; to edit the deployment descriptors in the components that make up the Enterprise application, see "Editing EJB Deployment Descriptors" on page 3-8, "Editing Web Application Deployment Descriptors" on page 3-10, or "Editing Resource Adapter Deployment Descriptors" on page 3-12.

To edit the Enterprise Application deployment descriptor:

1. Invoke the Administration Console in your browser:

   `http://`*`host`*`:`*`port`*`/console`

   where *`host`* refers to the name of the computer upon which WebLogic Server is running and *`port`* refers to the port number to which it is listening.

2. Click to expand the Deployments node in the left pane.

3. Click to expand the Applications node under the Deployments node.

4. Right-click the name of the Enterprise Application whose deployment descriptor you want to edit and choose Edit Application Descriptor from the drop-down menu. The Administration Console window appears in a new browser.

   The left pane contains a tree structure that lists all the elements in the `application.xml` file and the right pane contains a form for its descriptive elements, such as the display name and icon file names.

5. To edit an existing element in the `application.xml` deployment descriptor, follow these steps:

   a. Navigate the tree in the left pane, clicking on parent elements until you find the element you want to edit.

   b. Click the element. A form appears in the right pane that lists either its attributes or sub-elements.

   c. Edit the text in the form in the right pane.

   d. Click Apply.

6. To add a new element to the `application.xml` deployment descriptors:

a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to create.

b. Right-click the element and choose Configure a New *Element* from the drop-down menu.

c. Enter the element information in the form that appears in the right pane.

d. Click Create.

7. To delete an existing element from the `application.xml` deployment descriptor:

   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to delete.

   b. Right-click the element and chose Delete *Element* from the drop-down menu.

   c. Click Yes to confirm that you want to delete the element.

8. Once you make all your changes to the `application.xml` deployment descriptor, click the root element of the tree in the left pane. The root element is the either the name of the Enterprise application EAR archive file or the display name of the Enterprise application.

9. Click Validate if you want to ensure that the entries in the `application.xml` deployment descriptor are valid.

10. Click Persist to write your edits of the deployment descriptor files to disk in addition to WebLogic Server memory.

Packaging Web Applications

If your Web application is accessed by a programmatic Java client, see "Packaging Client Applications" on page 3-24, which describes how WebLogic server loads your application classes.

To stage and package a Web application:

1. Create a temporary staging directory anywhere on your hard drive. You can name this directory anything you want.
2. Copy all of your HTML files, JSP files, images, and any other files that these Web pages reference into the staging directory, maintaining the directory structure for referenced files. For example, if an HTML file has a tag such as `<img src="images/pic.gif">`, the `pic.gif` file must be in the `images` subdirectory beneath the HTML file.
3. Create `META-INF` and `WEB-INF/classes` subdirectories in the staging directory to hold deployment descriptors and compiled Java classes.
4. Copy or compile any servlet classes and helper classes into the `WEB-INF/classes` subdirectory.
5. Copy the home and remote interface classes for enterprise beans used by the servlets into the `WEB-INF/classes` subdirectory.
6. Copy JSP tag libraries into the `WEB-INF` subdirectory. (Tag libraries may be installed in a subdirectory beneath `WEB-INF`; the path to the `.tld` file is coded in the `.jsp` file.)
7. Set up your shell environment.

   On Windows NT, execute the `setenv.cmd` command, located in the directory *`server`*`\bin\setenv.cmd`, where *`server`* is the top-level directory in which WebLogic Server is installed.

   On UNIX, execute the `setenv.sh` command, located in the directory *`server`*`/bin/setenv.sh`, where *`server`* is the top-level directory in which WebLogic Server is installed.
8. Execute the following command to automatically generate the `web.xml` and `weblogic.xml` deployment descriptors in the `WEB-INF` subdirectory:

```
java weblogic.ant.taskdefs.war.DDInit staging-dir
```

where *staging-dir* refers to the staging directory.

For more information on the Java-based `DDInit` utility for generating deployment descriptors, see "Automatically Generating Deployment Descriptors" on page 3-5.

Alternatively, you can create the `web.xml` and `weblogic.xml` files manually in the `WEB-INF` subdirectory manually.

Note: See *Developing Web Applications for WebLogic Server* for detailed descriptions of the elements of the `web.xml` and `weblogic.xml` files.

9. Bundle the staging directory into a WAR file by executing a `jar` command such as:

   ```
   jar cvf myapp.war -C staging-dir
   ```

   The resulting WAR file can be added to an Enterprise application (EAR file) or deployed independently using the Administration Console or the `weblogic.Deployer` command-line utility.

   Note: Now that you have packaged your Web application, see *Deploying WebLogic Server Applications* for instructions on deploying applications in WebLogic Server.

Packaging Enterprise JavaBeans

You can stage one or more Enterprise JavaBeans (EJBs) in a directory and package them in an EJB JAR file. If your EJB is accessed by a programmatic Java client, see "Packaging Client Applications" on page 3-24 which describes how WebLogic Server loads your EJB classes.

Staging and Packaging EJBs

To stage and package an Enterprise JavaBean (EJB):

1. Create a temporary staging directory anywhere on your hard drive (for example, `c:\stagedir`).
2. Compile or copy the bean's Java classes into the staging directory.
3. Create a `META-INF` subdirectory in the staging directory.
4. Set up your shell environment.

   On Windows NT, execute the `setenv.cmd` command, located in the directory *`server`*`\bin\setenv.cmd`, where *`server`* is the top-level directory in which WebLogic Server is installed.

   On UNIX, execute the `setenv.sh` command, located in the directory *`server`*`/bin/setenv.sh`, where *`server`* is the top-level directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

5. If you are using EJB 1.1, e the following command to automatically generate the `ejb-jar.xml`, `weblogic-ejb-jar.xml`, and `weblogic-rdbms-cmp-jar-`*`bean_name`*`.xml` (if needed) deployment descriptors in the `META-INF` subdirectory:

   ```
   java weblogic.ant.taskdefs.ejb11.DDInit staging-dir
   ```

   where *`staging-dir`* refers to the staging directory. Use this utility for EJB 1.1.

   If you are creating EJB 2.0, execute:

   ```
   java weblogic.ant.taskdefs.ejb20.DDInit staging-dir
   ```

   For more information on the Java-based `DDInit` utility for generating deployment descriptors, see "Automatically Generating Deployment Descriptors" on page 3-5.

   Alternatively, you can create the EJB deployment descriptor files manually. Create an `ejb-jar.xml` and `weblogic-ejb-jar.xml` files in the `META-INF` subdirectory. If the bean is an entity bean with container-managed persistence, create a `weblogic-rdbms-cmp-jar–`*`bean_name`*`.xml` deployment descriptor in the `META-INF` directory with entries for the bean. Map the bean to this CMP deployment descriptor with a `<type-storage>` attribute in the `weblogic-ejb-jar.xml` file.

   Note: See *Programming WebLogic Enterprise JavaBeans* for help compiling enterprise beans and creating EJB deployment descriptors.

6. When all of the enterprise bean classes and deployment descriptors are set up in the staging directory, create the EJB JAR file with a `jar` command such as:

```
jar cvf jar-file.jar -C staging-dir
```

This command creates a JAR file that you can deploy on WebLogic Server.

The `-C staging-dir` option instructs the `jar` command to change to the `staging-dir` directory so that the directory paths recorded in the JAR file are relative to the directory where you staged the enterprise beans.

Enterprise beans require *container classes*, classes the WebLogic EJB compiler generates to allow the bean to deploy in a WebLogic Server. The WebLogic EJB compiler reads the deployment descriptors in the EJB JAR file to determine how to generate the classes. You can run the WebLogic EJB compiler on the JAR file before you deploy the beans, or you can let WebLogic Server run the compiler for you at deployment time. See *Programming WebLogic Enterprise JavaBeans* for help with the WebLogic EJB compiler.

Note: Now that you have packaged your EJB, see *Deploying WebLogic Server Applications* for instructions on deploying applications in WebLogic Server.

Using ejb-client.jar

WebLogic Server supports the use of `ejb-client.jar` files. Create an `ejb-client.jar` file by specifying this feature in the bean's `ejb-jar.xml` deployment descriptor file and then generating the `ejb-client.jar` file using `weblogic.ejbc`. An `ejb-client.jar` contains the class files that a client program needs to call the EJBs contained in the `ejb-jar` file. The files are the classes required to compile the client. If you specify this feature, WebLogic Server automatically creates the `ejb-client.jar`.

For more information, refer to "Packaging EJBs for the WebLogic Server Container" in *Programming WebLogic Enterprise JavaBeans.*

Packaging Resource Adapters

After you stage one or more resource adapters in a directory, you package them in a Java Archive (JAR). Before you package your resource adapters, be sure you read and understand the chapter entitled "WebLogic Server Application Classloading" in this guide, which describes how WebLogic Server loads classes.

To stage and package a resource adapter:

1. Create a temporary staging directory anywhere on your hard drive.
2. Compile or copy the resource adapter Java classes into the staging directory.
3. Create a JAR to store the resource adapter Java classes. Add this JAR to the top level of the staging directory.
4. Create a `META-INF` subdirectory in the staging directory.
5. Create an `ra.xml` deployment descriptor in the `META-INF` subdirectory and add entries for the resource adapter.

   Note: Refer to the following Sun Microsystems documentation for information on the `ra.xml` document type definition at:
   `http://java.sun.com/dtd/connector_1_0.dtd`

6. Create a `weblogic-ra.xml` deployment descriptor in the `META-INF` subdirectory and add entries for the resource adapter.

   Note: Refer to *Programming WebLogic Server J2EE Connectors* for information on the `weblogic-ra.xml` document type definition.

7. When the resource adapter classes and deployment descriptors are set up in the staging directory, you can create the RAR with a JAR command such as:

   `jar cvf` *`jar-file`*`.rar -C` *`staging-dir`*

   This command creates a RAR that you can deploy on a WebLogic Server or package in an enterprise application archive (EAR).

   The `-C` *`staging-dir`* option instructs the JAR command to change to the `staging-dir` directory so that the directory paths recorded in the JAR are relative to the directory where you staged the resource adapters.

Packaging Enterprise Applications

An Enterprise archive contains EJB and Web modules that are part of a related application. The EJB and Web modules are bundled together, along with the Enterprise Application deployment descriptor files, in another JAR file with an EAR extension.

Enterprise Applications Deployment Descriptor Files

The `META-INF` subdirectory in an EAR file contains an `application.xml` deployment descriptor provided by the application assembler; the format definition of this deployment descriptor is provided by Sun Microsystems. The `application.xml` deployment descriptor identifies the modules packaged in the EAR file.

You can find the DTD for the `application.xml` file at `http://java.sun.com/j2ee/dtds/application_1_2.dtd`.

Within `application.xml`, you define items such as the modules that make up your application and the security roles used within your application. The following is the `application.xml` file from the Pet Store example:

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

<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD
J2EE Application 1.2//EN'
'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  <display-name>estore</display-name>
  <description>Application description</description>
  <module>
    <web>
      <web-uri>petStore.war</web-uri>
      <context-root>estore</context-root>
    </web>
  </module>
  <module>
    <ejb>petStore_EJB.jar</ejb>
  </module>
  <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>
</application>
```

A supplemental deployment descriptor, `weblogic-application.xml` contains additional WebLogic-specific deployment information. This deployment descriptor is optional and is only needed if you want to configure *application scoping*.

Application scoping refers to configuring resources for a particular enterprise application rather than for an entire WebLogic Server configuration. Examples of resources include the XML parser used by an application, the EJB entity cache, the JDBC connection pool, and so on. The main advantage of application scoping is that it isolates the resources for a given application to the application itself.

Another advantage of using application scoping is that by associating the resources with the EAR file, you can run this EAR file on another instance of WebLogic Server without having to configure the resources for that server.

Refer to "weblogic-application.xml Deployment Descriptor Elements" in Appendix A, "Application Deployment Descriptor Elements," for `weblogic-application.xml` deployment descriptor elements.

Packaging Enterprise Applications: Main Steps

If your enterprise application is accessed by a programmatic Java client, see "Packaging Client Applications" on page 3-24, which describes how WebLogic Server loads your enterprise application classes.

To stage and package an Enterprise application:

1. Create a temporary staging directory anywhere on your hard drive.
2. Copy the Web archives (WAR files) and EJB archives (JAR files) into the staging directory.
3. Create a `META-INF` subdirectory in the staging directory.
4. Set up your shell environment.

On Windows NT, execute the `setenv.cmd` command, located in the directory *`server`*`\bin\`*`setenv.cmd`*, where *`server`* is the top-level directory in which WebLogic Server is installed.

On UNIX, execute the `setenv.sh` command, located in the directory *`server`*`/bin/`*`setenv.sh`*, where *`server`* is the directory in which WebLogic Server is installed.

5. Execute the following command to automatically generate the `application.xml` deployment descriptor in the `META-INF` subdirectory:

   ```
   java weblogic.ant.taskdefs.ear.DDInit staging-dir
   ```

   where *`staging-dir`* refers to the staging directory.

   For more information on the Java-based `DDInit` utility for generating deployment descriptors, see "Automatically Generating Deployment Descriptors" on page 3-5.

   Alternatively, you can create the `application.xml` file automatically in the `META-INF` directory. See Appendix A, "Application Deployment Descriptor Elements," for detailed information about the elements in this file.

6. Optionally create the `weblogic-application.xml` file manually in the `META-INF` directory, as described in Appendix A, "Application Deployment Descriptor Elements."

7. Create the Enterprise Archive (EAR file) for the application, using a `jar` command such as:

   ```
   jar cvf application.ear -C staging-dir
   ```

   The resulting EAR file can be deployed using the Administration Console or the `weblogic.Deployer` command-line utility.

   Note: Now that you have packaged your enterprise application, see *Deploying WebLogic Server Applications* for instructions on deploying applications in WebLogic Server.

Packaging Client Applications

Although not required for WebLogic Server applications, J2EE includes a standard for deploying client applications. A J2EE client application module is packaged in a JAR file. This JAR file contains the Java classes that execute in the client JVM (Java Virtual Machine) and deployment descriptors that describe EJBs (Enterprise JavaBeans) and other WebLogic Server resources used by the client.

A de-facto standard deployment descriptor `application-client.xml` from Sun is used for J2EE clients and a supplemental deployment descriptor contains additional WebLogic-specific deployment information.

Note: See "application-client.xml Deployment Descriptor Elements" in Appendix B, "Client Application Deployment Descriptor Elements," for help with these deployment descriptors.

Executing a Client Application in an EAR File

In order to simplify distribution of an application, J2EE defines a way to include client-side components in an EAR file, along with the server-side modules that are used by WebLogic Server. This enables both the server-side and client-side components to be distributed as a single unit.

The client JVM must be able to locate the Java classes you create for your application and any Java classes your application depends upon, including WebLogic Server classes. You stage a client application by copying all of the required files on the client into a directory and bundling the directory in a JAR file. The top level of the client application directory can have a batch file or script to start the application. Create a `classes` subdirectory to hold Java classes and JAR files, and add them to the client `Class-Path` in the startup script. You may also want to package a Java Runtime Environment (JRE) with a Java client application.

Note: The use of the `Class-Path` manifest entries in client component JARs is not portable, because it has not yet been addressed by the J2EE standard.

The `Main-Class` attribute of the JAR file manifest defines the main class for the client application. The client typically uses `java:/comp/env` JNDI lookups to execute the `Main-Class` attribute. As a deployer, you must provide runtime values for the JNDI lookup entries and populate the component local JNDI tree before calling the client's `Main-Class` attribute. You define JNDI lookup entries in the client deployment descriptor. (Refer to "Client Application Deployment Descriptor Elements.")

You use `weblogic.ClientDeployer` to extract the client-side JAR file from a J2EE EAR file, creating a deployable JAR file. The `weblogic.ClientDeployer` class is executed on the Java command line with the following syntax:

```
java weblogic.ClientDeployer ear-file client
```

The *`ear-file`* argument is an expanded directory (or Java archive file with a `.ear` extension) that contains one or more client application JAR files.

For example:

```
java weblogic.ClientDeployer app.ear myclient
```

where `app.ear` is the EAR file that contains a J2EE client packaged in `myclient.jar`.

Once the client-side JAR file is extracted from the EAR file, use the `weblogic.j2eeclient.Main` utility to bootstrap the client-side application and point it to a WebLogic Server instance as follows:

```
java weblogic.j2eeclient.Main clientjar URL [application args]
```

For example

```
java weblogic.j2eeclient.Main helloWorld.jar t3://localhost:7001 Greetings
```

Special Considerations for Deploying J2EE Client Applications

The following is a list of special considerations for deploying J2EE client applications:

- Name the WebLogic Server client deployment file using the suffix `.runtime.xml`.
- The `weblogic.ClientDeployer` class is responsible for generating and adding a `client.properties` file to the client JAR file. A separate program,

`weblogic.j2eeclient.Main`, creates a local client JNDI context and runs the client from the entry point named in the client manifest file.

Note: To run the J2EE client application using `weblogic.ClientDeployer`, you need the `weblogic.j2eeclient.Main` class (located in the `weblogic.jar` file).

- If a resource mentioned by the `application-client.xml` file is one of the following types, the `weblogic.j2eeclient.Main` class attempts to bind it from the global JNDI tree on the server to `java:comp/env/`:

  `ejb-ref`

  `javax.jms.QueueConnectionFactory`

  `javax.jms.TopicConnectionFactory`

  `javax.mail.Session`

  `javax.sql.DataSource`

- The `weblogic.j2eeclient.Main` class binds `UserTransaction` to `java:comp/UserTransaction`.
- The rest of the client environment is bound from the `client.properties` file created by the `weblogic.ClientDeployer` class into `java:comp/env/`. The `weblogic.j2eeclient.Main` class emits error messages for missing or incomplete bindings.
- The `<res-auth>` tag in the application deployment file is currently ignored and should be entered as `Application`. We do not currently support form-based authentication.

Note: For more information on deploying, refer to *Deploying WebLogic Server Applications*.

Packaging J2EE Applications Using Apache Ant

The topics in this section discuss building and packaging J2EE applications using Apache Ant, an extensible Java-based tool. Ant is similar to the `make` command but is designed for building Java applications. Ant libraries are bundled with WebLogic Server to make it easier for our customers to build Java applications out of the box.

Developers write Ant build scripts using eXtensible Markup Language (XML). XML tags define the targets to build, dependencies among targets, and tasks to execute in order to build the targets.

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

Packaging J2EE Deployment Units

As previously discussed, J2EE applications are packaged as JAR files containing a specific file extension depending on the component type:

- EJBs are packaged as JAR files.
- Web Applications are packaged as WAR files.
- Resource Adapters are packaged as RAR files.
- Enterprise Applications are packaged as EAR files.

These components are structured according to the J2EE specifications. In addition to the standard XML deployment descriptors, components may also be packaged with WebLogic Server-specific XML deployment descriptors.

Ant provides tasks that make the construction of these JAR files easier. In addition to the features of the JAR command, Ant provides specific tasks for building EAR and WAR files. Using Ant, you can specify the pathname as it appears in the JAR archive, which may differ from the original path in the file system. This ability is useful for packaging deployment descriptors (in which J2EE specifies an exact location in the

archive), which may not correspond to the location in your source tree. See the Apache Ant online documentation pertaining to the `ZipFileSet` command for related information.

The following listing shows:

Listing 3-1 WAR Task Example

```
<war warfile="cookie.war" webxml="web.xml"
manifest="manifest.txt">

   <zipfileset dir="." prefix="WEB-INF" includes="weblogic.xml"/>

   <zipfileset dir="." prefix="images" includes="*.gif,*.jpg"/>

   <classes dir="classes" includes="**/CookieCounter.class"/>

   <fileset dir="." includes="*.jsp,*.html">

   </fileset>

</war>
```

Packaging J2EE deployment units requires the following steps:

1. Specify the standard XML deployment descriptor using the `webxml` parameter.
2. The `war` task automatically maps XML deployment descriptor to the standard name in the WAR archive `WEB-INF/web.xml`.
3. Apache Ant stores the `manifest` file, specified using the `manifest` parameter, under the standard name `META-INF/MANIFEST.MF`.
4. Use the Apache Ant `ZipFileSet` command to define a set of files (in this case, just the WebLogic Server-specific deployment descriptor `weblogic.xml`) that should be stored in the `WEB-INF` directory.
5. Use a second `ZipFileSet` command to package all the images in an `images` directory.
6. The `classes` tag packages servlet classes in the `WEB-INF/classes` directory.
7. Finally, add all the `.jsp` and `.html` files from the current directory to the archive.

You can achieve the same result by staging the files in a directory that directly corresponds to the structure of the WAR file and creating a JAR file from that directory. Using special features of the Ant JAR tasks eliminates the need to copy files into a specific directory hierarchy.

The following example builds a Web application and an EJB, and then packages them together in an EAR file:

Listing 3-2 Packaging Example

```
<project name="app" default="app.ear">
   <property name="wlhome" value="/bea/wlserver6.1"/>
   <property name="srcdir" value="/bea/myproject/src"/>
   <property name="appdir" value="/bea/myproject/config/mydomain/applications"/>
   <target name="timer.war">
       <mkdir dir="classes"/>
      <javac srcdir="${srcdir}" destdir="classes" includes="myapp/j2ee/timer/*.java"/>
        <war warfile="timer.war" webxml="timer/web.xml"
manifest="timer/manifest.txt">
          <classes dir="classes" includes="**/TimerServlet.class"/>
       </war>
   </target>
   <target name="trader.jar">
   <mkdir dir="classes"/>
   <javac srcdir="${srcdir}" destdir="classes" includes="myapp/j2ee/trader/*.java"/>
   <jar jarfile="trader0.jar" manifest="trader/manifest.txt">
      <zipfileset dir="trader" prefix="META-INF" includes="*ejb-jar.xml"/>
      <fileset dir="classes" includes="**/Trade*.class"/>
      </jar>
      <ejbc source="trader0.jar" target="trader.jar"/>
   </target>
```

```
   <target name="app.ear" depends="trader.jar, timer.war">
      <jar jarfile="app.ear">
         <zipfileset dir="." prefix="META-INF" includes="application.xml"/>
         <fileset dir="." includes="trader.jar, timer.war"/>
      </jar>
   </target>
   <target name="deploy" depends="app.ear">
      <copy file="app.ear" todir="${appdir}/>
   </target>
</project>
```

Running Ant

BEA provides a simple script to run Ant in the `server/bin` directory. By default, Ant loads the `build.xml` build file, but you can override this using the `-f` flag. Use the following command to build and deploy an application using the build script shown above:

ant -f yourbuildscript.xml

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 component of the Java language. A classloader is a part of the Java virtual machine (JVM) that loads classes into memory; it is the class 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 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 to see if the requested class has already been loaded. This class verification improves performance in that the cached memory copy is used instead of repeated loading of a class from disk. If the class is not found in memory, 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 following section discusses this in more detail.

PreferWebInfClasses Element

The `WebAppComponentMBean` contains a `PreferWebInfClasses` element. By default, this element is set to `False`. When you set this element to `True`, this 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.

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

Listing 4-1 PreferWebInfClasses 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 components 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 reinstantiated.

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. This is the topic of the next section.

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 Adapter RAR files and classloading, see "About Resource Adapter Classes."

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

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

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 classloaders, no sibling or friend concepts exist. Application code only has visibility to classes loaded by the classloader associated with the application (or component) and classes that are loaded by classloaders that are ancestors of the application (or component) classloader. This allows WebLogic Server to host multiple isolated applications within the same JVM.

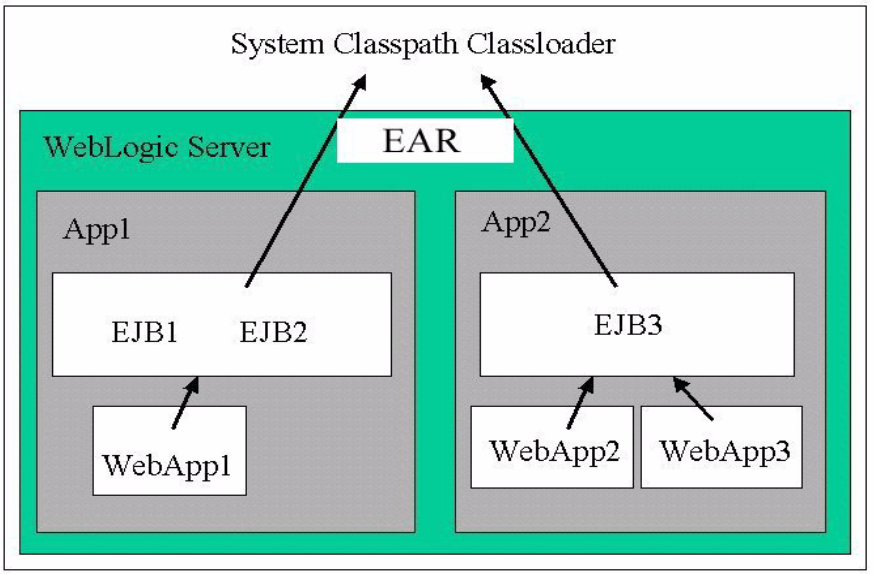
Application Classloader Hierarchy

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 beans 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 4-15.

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.

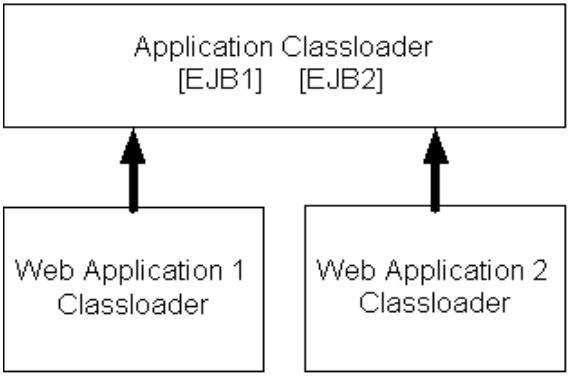
Custom Module Classloader Hierarchies

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 on 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 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, "Application Deployment Descriptor Elements."

The following is an example of what a classloader declaration would look like:

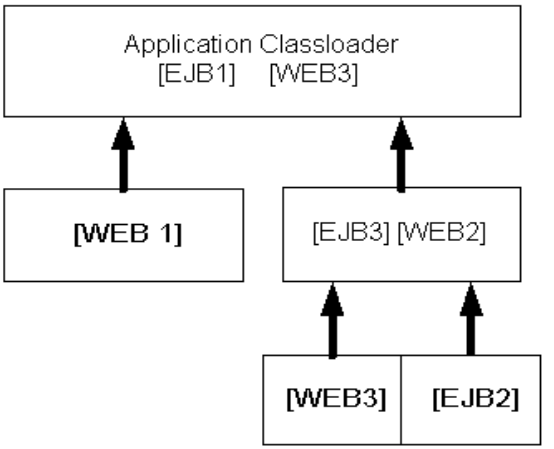
Listing 4-3 Example Classloader Declaration

```
<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>ejb1.jar</module-uri>
            </module-ref>
            <module-ref>
                <module-uri>web2.war</module-uri>
            </module-uri>
            <classloader>
                <module-ref>
                    <module-uri>web4.war</module-uri>
                </module-ref>
            </classloader>
            <classloader>
                <module-ref>
                    <module-uri>ejb2.jar</module-uri>
                </module-ref>
            </classloader>
        </classloader>
    </classloader>
```

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

The purpose of this feature is to provide you with better control over what is reloadable and provide inter-module class visibility. This is primarily intended to be a developer feature. It is useful for iterative development, but the reloading aspect of this feature is not recommended for production use, since it is possible to corrupt a running application if an update includes invalid elements. Custom classloader arrangements for namespace separation and class visibility are acceptable for production use. However, programmers should be aware that the J2EE specifications say that applications should not depend on any given classloader organization.

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

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

Servlet Reloading Disabled

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

Nesting Depth

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

Module Types

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

Duplicate Entries

Duplicate entries lead to a deployment exception.

Interfaces

With our standard classloader hierarchy, the interfaces for EJB are available to all modules in the application. This means that other modules can invoke on an EJB, even though they do not include the interface classes in their own module. This is possible since 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

It is important to be aware that the classloader switch required for reloading is not atomic across modules. In fact, updates to applications are in general not atomic. For this reason, it is possible that different in-flight operations might end up accessing different versions of classes depending on timing.

Development Use Only

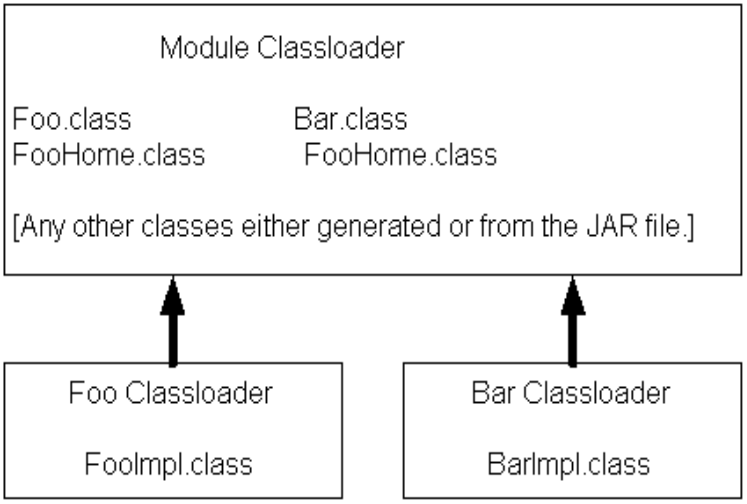
This feature is intended for development use. Since updates are not atomic, this feature is not suitable for production use.

Individual EJB Classloader for Implementation Classes

WebLogic Server allows you to reload individual EJB modules without forcing other modules to be reloaded 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.

Since 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 an incremental update (partial upgrade), use the following command line:

Listing 4-4

```
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

```
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 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 our 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. Since 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 components together as an EAR file to enable fast RMI calls and use of the EJB 2.0 local interfaces.

Resolving Class References Between Components 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 component 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 component. This section describes how WebLogic Server uses multiple classloaders so that you can stage your applications successfully.

About Resource Adapter Classes

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 components (for example, an EJB or Web application), you must bundle these classes in the corresponding component'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, making them visible to all components within the EAR.

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 component 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 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

CHAPTER

5 Programming Topics

The following sections contain information about programming in the WebLogic Server environment, including descriptions of useful WebLogic Server facilities and advice about using various programming techniques:

Logging Messages

Each WebLogic Server instance has a log file that contains messages generated from that server. Your applications can write messages to the log file using internationalization services that access localized message catalogs. If localization is not required, you can use the `weblogic.logging.NonCatalogLogger` class to write messages to the log. This class can also be used in client applications to write messages in a client-side log file.

For more information, refer to the Using WebLogic Logging Services guide.

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 components it hosts. To obtain the greatest advantage from WebLogic Server's architecture, construct your application components created according to the standard J2EE APIs.

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

- 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 components 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 components. 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.

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.

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; so 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 components 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 components 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 component 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`	The protocol to use to retrieve email. Example: `mail.store.protocol=imap`	The bundled JavaMail library has support for IMAP.
`mail.transport.protocol`	The protocol to use to send email. Example: `mail.transport.protocol=smtp`	The bundled JavaMail library has support for SMTP.
`mail.host`	The name of the mail host machine. Example: `mail.host=mailserver`	The default is the local machine.
`mail.user`	The name of the default user for retrieving email. Example: `mail.user=postmaster`	The default is the value of the `user.name` Java system property.

Property	Description	Default

`mail.`*`protocol`*`.host`	The 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`
		The value of the `mail.host` property.

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

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

You can override any properties set in the Mail Session in your code by creating a `Properties` object containing the properties you want to override. Then, after you lookup the Mail Session object in JNDI, call the `Session.getInstance()` method with your `Properties` to get a customized Session.

Sending Messages with JavaMail

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

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 component:

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 *Using WebLogic Server Clusters* for more information.

EJBs deployed in a WebLogic Server cluster have certain restrictions based on EJB type. See "The WebLogic Server EJB Container" 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. "weblogic-ejb-jar.xml Deployment Descriptors" in *Programming WebLogic Enterprise JavaBeans* describes the XML deployment elements relevant for clustering.

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

CHAPTER

A Application Deployment Descriptor Elements

The following sections describe deployment descriptors for J2EE applications on WebLogic Server. Two deployment descriptors are required: a J2EE standard deployment descriptor named `application.xml`, and a WebLogic-specific application deployment descriptor named `weblogic-application.xml`. The `weblogic-application.xml` file is optional if you are not using any WebLogic Server extensions.

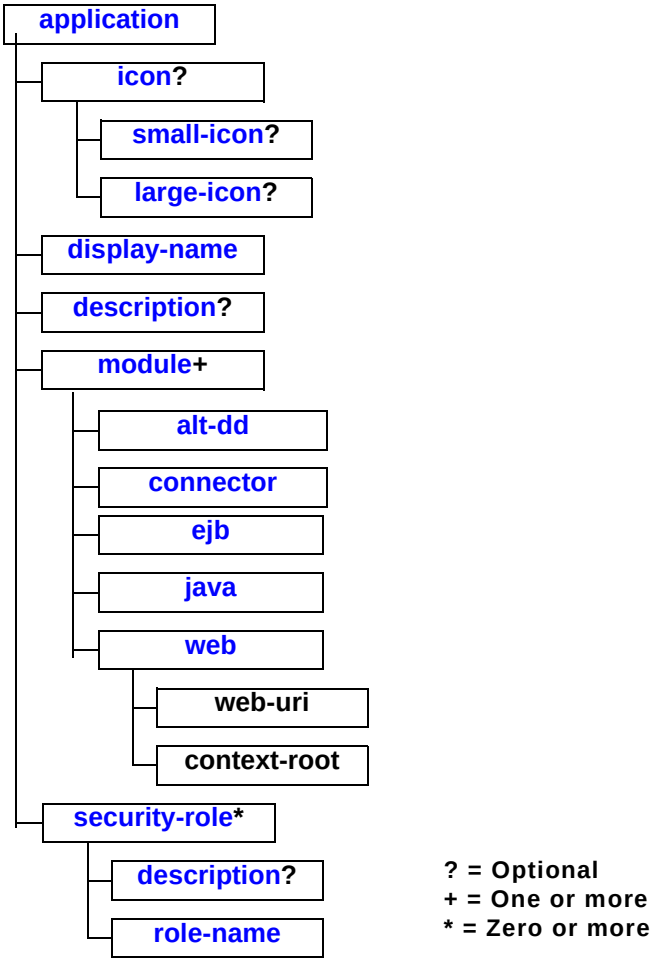
application.xml Deployment Descriptor Elements

The following sections describe the `application.xml` file.

The `application.xml` file is the deployment descriptor for Enterprise Application Archives. 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">
```

The following diagram summarizes the structure of the `application.xml` deployment descriptor.



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

application

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

icon

Optional. The `icon` element 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.

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.

display-name

Optional. The `display-name` element specifies the application display name, a short name that is intended to be displayed by GUI tools.

description

The optional description element provides descriptive text about the application.

module

The `application.xml` deployment descriptor contains one `module` element for each module in the Enterprise Archive file. Each `module` element contains an `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.

alt-dd

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 component specification.

connector

Specifies the URI of a resource adapter (connector) archive file, relative to the top level of the application package.

ejb

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>
```

java

Defines a client application module in the application file.

Example:

```
<java>client_app.jar</java>
```

web

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 having 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 `security-role` element 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.

description

Optional. Text description of the security role.

role-name

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 7.1.0//EN"

"http://www.bea.com/servers/wls710/dtd/weblogic-application_2_0.d
td";>
```

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.

ejb

Optional. The `ejb` element contains information that is specific to the EJB modules that are part of a WebLogic application. Currently, one can use the `ejb` element to specify one or more application level caches that can be used by the application's entity beans.

entity-cache

One 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>
```

entity-cache-name

The `entity-cache-name` element specifies a unique name for an entity bean cache. The name must be unique within an ear file and may not be the empty string.

Example:

```
<entity-cache-name>ExclusiveCache</entity-cache-name>
```

max-beans-in-cache

Optional. The `max-beans-in-cache` element specifies the maximum number of entity beans that are allowed in the cache. If the limit is reached, beans may be passivated. If `0` is specified, then there is no limit. This mechanism does not take into account the actual amount of memory that different entity beans require.

Default Value: `1000`

max-cache-size

The `max-cache-size` element is used to specify a limit on the size of an entity cache in terms of memory size—expressed either in terms of bytes or megabytes. A bean provider should provide an estimate of the average size of a bean in the `weblogic-ejb-jar.xml` descriptor if the bean uses a cache that specifies its maximum size using the `max-cache-size` element. By default, a bean is assumed to have an average size of 100 bytes.

- `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.

read-timeout-seconds

Optional. The `read-timeout-seconds` element specifies the number of seconds between `ejbLoad` calls on a read-only entity bean. If `read-timeout-seconds` is set to `0`, `ejbLoad` will only be called when the bean is brought into the cache.

caching-strategy

Optional. The `caching-strategy` element specifies the general strategy that the EJB container uses to manage entity bean instances in a particular application level cache. A cache buffers entity bean instances in memory and associates them with their primary key value.

The `caching-strategy` element can 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>
```

start-mdbs-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.
```

xml

Optional. The `xml` element contains information about parsers and entity mappings for XML processing that is specific to this application.

parser-factory

Optional. The `parser-factory` element contains three elements: `saxparser-factory?`, `document-builder-factory?`, and `transformer-factory?`.

saxparser-factory

Optional. The `saxparser-factory` element allows you to set the SAXParser Factory for the XML parsing required in this application only. This element determines the factory to be used for SAX style parsing. If you do not specify the `saxparser-factory` element setting, the configured SAXParser Factory style in the Server XML Registry is used.

Default Value: Server XML Registry setting

document-builder-factory

Optional. The `document-builder-factory` element 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. The `transformer-factory` element 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

Zero or more. The `entity-mapping` element is used to specify 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.

entity-mapping-name

The `entity-mapping-name` element specifies the name for this entity mapping.

public-id

Optional. The `public-id` element specifies the public ID of the mapped entity.

system-id

Optional. The `system-id` element specifies the system ID of the mapped entity.

entity-uri

Optional. The `entity-uri` element 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. The `cache-timeout-interval` element allows you to specify the integer value in seconds.

security

Optional. The `security` element specifies security information for the application.

- `realm-name`—Optional. The `realm-name` element names a security realm that will be used by the application. If not specified, the system default realm is used.

jdbc-connection-pool

Zero or more. The `jdbc-connection-pool` element specifies an application-scoped JDBC connection pool.u

data-source-name

The `data-source-name` element specifies the JNDI name in the application-specific JNDI tree.

connection-factory

The `connection-factory` element defines the number of physical database connections to create when the pool is initialized. The default value is `1`.

factory-name

The `factory-name` element specifies the name of a `JDBCDataSourceFactoryMBean` in the `config.xml` file.

connection-properties

Optional. The `connection-properties` element 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

pool-params

Optional. The `pool-params` element defines parameters that affect the behavior of the pool.

size-params

Optional. The `size-params` element 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`—
- `highest-num-waiters`—
- `highest-num-available`—
- `profiling-enabled`—
- `cache-profiling-threshold`—
- `cache-size`—
- `parameter-logging-enabled`—
- `max-parameter-length`—
- `acl-name`—The ACL used to control access to this connection pool.

xa-params

Optional. The `xa-params` element 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`—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.
- `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.

login-delay-seconds

Optional. Integer value. The `login-delay-seconds` element 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. The `leak-profiling-enabled` element 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.

connection-check-params

Optional. The `connection-check-params` element 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.
- `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.

driver-params

Optional. The `driver-params` element sets behavior on WebLogic Server drivers.

statement

Optional.

- `profiling-enabled`—Optional. `profiling-enabled` boolean. The `profiling-enabled` element enables the running of JDBC SQL round-trip profiling. When enabled, SQL statement text, execution time, and other metrics are stored externally 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`.

prepared-statement

Optional. `profiling-enabled` boolean. The `prepared-statement` element 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.

row-prefetch-enabled

Optional

row-prefetch-size

Optional

stream-chunk-size

Optional

acl-name

Optional

application-param

Zero or more. The `application-param` element defines various parameters that affect container behavior. These parameters are as follows:

- `webapp.encoding.usevmdefault`
- `webapp.encoding.default`
- `webapp.getrealpath.accept_context_path`

classloader-structure

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 in this structure.

Example:

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

module-ref

Zero or more.

module-uri

classloader-structure

Zero or more.

listener

The listener element is used to register user defined application lifecycle listeners. These are classes that extend the abstract base class `weblogic.application.ApplicationLifecycleListener`.

listener-class

The `listener-class` element is the name of the users implementation of `ApplicationLifecycleListener`.

listener-uri

Optional. The `listener-uri` is 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

Use the `startup` element to register user-defined startup classes.

startup-class

Use the `startup-class` element to define the name of the class to be run when the application is being deployed.

startup-uri

Optional. Use the `startup-uri` element to define 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 `shutdown` element is used to register user defined shutdown classes.

shutdown-class

Use the `shutdown-class` element to define the name of the class to be run when the application is undeployed.

shutdown-uri

Optional. The `shutdown-uri` element is used to define 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.

CHAPTER

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 components (Web Applications, EJBs, Connectors). You configure these server-side components using the `application.xml` deployment descriptor, discussed in Appendix A, "Application Deployment Descriptor Elements."

However, it is also possible to include a client component (a JAR file) in an EAR file. This JAR file is only used on the client side; you configure this client component using the `client-application.xml` deployment descriptor. This scheme makes it possible to package both client and server side components 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).

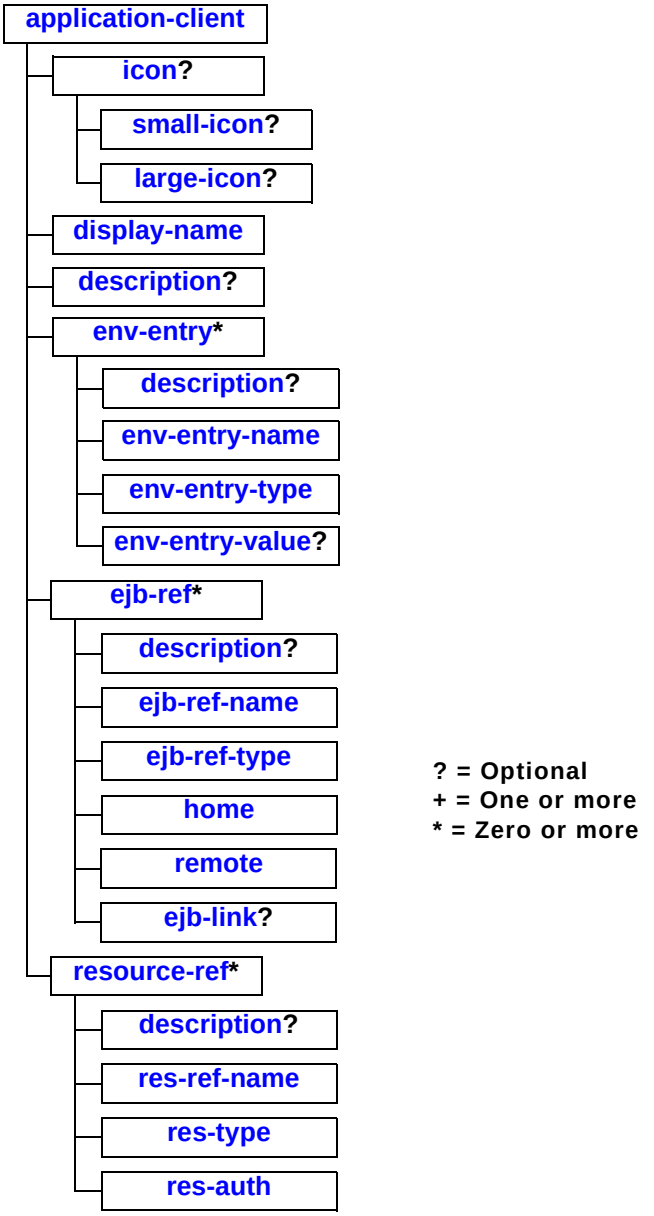
For client-side components, two deployment descriptors are required: a J2EE standard deployment descriptor, `application-client.xml`, and a WebLogic-specific runtime deployment descriptor with a name derived from the client application JAR file.

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 diagram summarizes the structure of the `application-client.xml` deployment descriptor.



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 components and other resources used by the client application.

The elements within the `application-client` element are described in the following sections.

icon

Optional. The `icon` element 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.

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.

display-name

The `display-name` element 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.

env-entry

The `env-entry` element contains the declaration of a client application's environment entries.

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`.

ejb-ref

The `ejb-ref` element is used for the declaration of a reference to an EJB referenced in the client application.

description

Optional. The `description` element provides a description of the referenced EJB.

ejb-ref-name

The `ejb-ref-name` element contains the name of the referenced EJB. Typically the name is prefixed by `ejb/`, such as `ejb/Deposit`.

ejb-ref-type

The `ejb-ref-type` element contains the expected type of the referenced EJB, either `Session` or `Entity`.

home

The `home` element contains the fully-qualified name of the referenced EJB's home interface.

remote

The `remote` element contains the fully-qualified name of the referenced EJB's remote interface.

ejb-link

The `ejb-link` element 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.

resource-ref

The `resource-ref` element contains a declaration of the client application's reference to an external resource.

description

Optional. The `description` element contains a description of the referenced external resource.

res-ref-name

The `res-ref-name` element 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

The `res-type` element 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

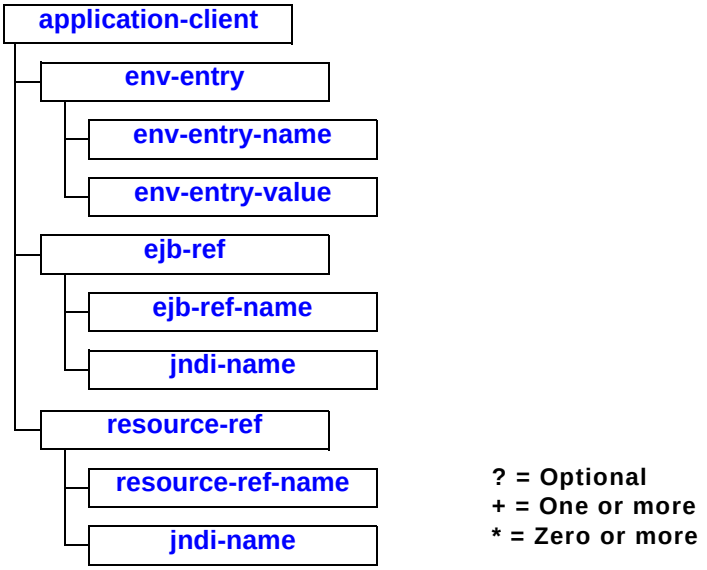
The `res-auth` element specifies whether the EJB code signs on programmatically to the resource manager, or whether the Container will sign on to the resource manager on behalf of the EJB. In the latter case, the Container uses information that is supplied by the Deployer. The res-auth element can have 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`.

The following diagram shows the structure of the elements in the run-time deployment descriptor.



application-client

The `application-client` element is the root element of a WebLogic-specific run-time client deployment descriptor.

env-entry

The `env-entry` element specifies values for environment entries declared in the deployment descriptor.

env-entry-name

The `env-entry-name` element contains the name of an application client's environment entry.

Example:

```
<env-entry-name>EmployeeAppDB</env-entry-name>
```

env-entry-value

The `env-entry-value` element contains the value of an application client's environment entry. The value must be a string valid for the constructor of the specified type that takes a single string parameter.

ejb-ref

The `ejb-ref` element specifies the JNDI name for a declared EJB reference in the deployment descriptor.

ejb-ref-name

The `ejb-ref-name` element 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

The `jndi-name` element specifies the JNDI name for the EJB.

resource-ref

The `resource-ref` element 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>
```

resource-ref-name

The `res-ref-name` element 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

The `jndi-name` element specifies the JNDI name for the resource.

Index

D

L

M

P

R

S

T

W

X