BEA WebLogic Server™

Developing WebLogic Server Applications

BEA WebLogic Server Version 6.1
Document Date: June 24, 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Developing WebLogic Server Applications

Part Number	Document Date	Software Version
N/A	June 24, 2002	BEA WebLogic Server Version 6.1

Contents

2. Developing WebLogic Server J2EE Applications

3. Packaging WebLogic Server J2EE Applications

4. Programming Topics

A. application.xml 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 J2EE Applications,” describes components of WebLogic Server applications.
- Chapter 2, “Developing WebLogic Server J2EE Applications,” outlines high-level procedures for creating WebLogic Server applications and helps Java programmers establish their programming environment.
- Chapter 3, “Packaging WebLogic Server J2EE Applications,” describes how to bundle WebLogic Server components and applications in standard JAR files for distribution and deployment.
- Chapter 4, “Programming Topics,” covers general WebLogic Server application programming issues, such as logging messages and using threads.
- Appendix A, “application.xml Deployment Descriptor Elements,” is a reference for the standard J2EE Enterprise application deployment descriptor, `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/docs61/ejb/index.html
- *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs61/jsp/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html
- *Programming WebLogic JDBC* at http://e-docs.bea.com/wls/docs61/jdbc/index.html
- *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs61/webServices/index.html
- *Programming WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/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.deploy [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 J2EE Applications

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

- What Are WebLogic Server J2EE Applications and Components?
- Web Application Components
- Enterprise JavaBean Components
- WebLogic Server Components
- Connector Component
- Enterprise Applications
- Client Applications

What Are WebLogic Server J2EE Applications and Components?

A BEA WebLogic Server™ application is an application composed of one or many J2EE components that runs on WebLogic Server. They can include the following components:

- Web components—HTML pages, servlets, JavaServer Pages, and related files
- EJB components—entity beans, session beans, and message-driven beans
- WebLogic Server components—startup and shutdown classes
- Connector component—resource adapters

Web designers, application developers, and application assemblers create applications and their components by using J2EE technologies such as JavaServer Pages, servlets, Enterprise JavaBeans, and 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, 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.deploy` command-line utility to upload JAR files to the target WebLogic Servers.

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

J2EE Platform

WebLogic Server contains Java 2 Platform, Enterprise Edition (J2EE) technologies. 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.

WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality

BEA WebLogic Server 6.1 is the first e-commerce transaction platform to implement advanced J2EE 1.3 features. To comply with the rules governing J2EE, BEA Systems provides two separate downloads: one with J2EE 1.3 features enabled, and one that is limited to J2EE 1.2 features only. Both downloads offer the same container and differ only in the APIs that are available.

Note: Your CLASSPATH setting for compiling J2EE components depends on whether you want to create components that are completely J2EE 1.2-compliant or components that contain J2EE 1.3 features. For detailed information, see "Setting the Classpath for Compiling" on page 2-18.

WebLogic Server 6.1 with J2EE 1.2 Plus Additional J2EE 1.3 Features

With this download, WebLogic Server defaults to running with J2EE 1.3 features enabled. These features include EJB 2.0, JSP 1.2, Servlet 2.3, and J2EE Connector Architecture 1.0. When you run WebLogic Server 6.1 with J2EE 1.3 features enabled, J2EE 1.2 applications are still fully supported. The J2EE 1.3 feature implementations use non-final versions of the appropriate API specifications. Therefore, application

code developed for BEA WebLogic Server 6.1 that uses the new features of J2EE 1.3 may be incompatible with the J2EE 1.3 platform supported in future releases of BEA WebLogic Server.

WebLogic Server 6.1 with J2EE 1.2 Certification

With this download, WebLogic Server defaults to running with J2EE 1.3 features disabled and is fully compliant with the J2EE 1.2 specification and regulations.

Web Application Components

A Web archive contains the files that make up a Web application. A `.war` file is deployed as a unit on one or more WebLogic Servers.

A Web archive on WebLogic Server always includes the following files:

- at least one servlet or JSP page, 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 archive might also include HTML/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

JSP pages are Web pages coded with an extended HTML that makes it possible to embed Java code in a Web page. JSP pages can call custom Java classes, called taglibs, using HTML-like tags. The WebLogic JSP compiler, `weblogic.jspc`, translates JSP pages into servlets. WebLogic Server automatically compiles JSP pages if the servlet class file is not present or is older than the JSP source file.

You can also precompile JSP pages and package the servlet class in the Web Archive to avoid compiling in the server. Servlets and JSP pages may depend upon additional helper classes that must also be deployed with the Web application.

Web Application Directory Structure

Web application components are assembled in a directory in order to stage the `.war` file for the `jar` command. HTML pages, JSP pages, 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 JSP pages.

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

For More Information on Web Application Components

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

- *Programming WebLogic Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs61/jsp/index.html
- *Writing JSP Extensions* at http://e-docs.bea.com/wls/docs61/taglib/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/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 without the advantages of running in the EJB container.

Enterprise beans are bundled into a JAR file 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.ejbc`, to generate the stub and skeleton classes that allow an enterprise bean to be executed remotely.

WebLogic stubs and skeletons can also contain support for WebLogic clusters, which enable load-balancing and failover for enterprise beans. You can run `weblogic.ejbc` to generate the stub and skeleton classes and add them to the EJB JAR file, or WebLogic Server can create them by running the compiler at deployment time.

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/docs61/ejb/index.html.

WebLogic Server Components

The WebLogic Server components are startup and shutdown classes, Java classes that execute when deployed or at shutdown time, respectively.

Startup classes can be RMI classes that register themselves in the WebLogic Server naming tree or any other Java class that can be executed in WebLogic Server. Startup classes can be used to implement new services in WebLogic Server. You could create a startup class that provides access to a legacy application or a real-time feed, for example.

Shutdown classes execute when WebLogic Server shuts down and are usually used to free resources obtained by startup classes.

Startup and shutdown classes can be configured in WebLogic Server from the Administration Console. The Java class must be in the server's CLASSPATH.

Connector Component

The central component within the WebLogic J2EE Connector architecture is the resource adapter, which serves as the "connector." The Connector architecture enables both Enterprise Information Systems (EISs) 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.

When a resource adapter is deployed in the WebLogic Server environment, it enables the development of robust J2EE applications that now have access to 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`) 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.

For help configuring and deploying resource adapters, see *Programming the WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/jconnector/index.html.

Enterprise Applications

An enterprise J2EE application contains both Web and EJB components, deployment descriptors, and archive files. An Enterprise Archive (`.ear`) file contains the Web archives and EJB archives. 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.deploy` command line utility to deploy an `.ear` file on one or more WebLogic Servers in a domain.

Client Applications

Client-side applications written in Java 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.

Client applications use WebLogic Server components indirectly, using HTTP requests or RMI requests. The components actually execute in WebLogic Server, not in the client.

To execute a WebLogic Server Java client, the client computer needs the `weblogic.jar` file, `weblogic_sp.jar` file (if you are using a Service Pack version of WebLogic Server), the remote interfaces for any RMI classes and enterprise beans on WebLogic Server, and the client application classes.

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 `weblogic.jar` and `weblogic_sp.jar` files.

WebLogic Server also supports J2EE client applications (as opposed to simple Java programs) that are packaged in a JAR file with a standard XML deployment descriptor (`client-application.xml`) and a WebLogic-specific deployment descriptor. The `weblogic.ClientDeployer` command line utility is executed on the client computer to package a client application to this specification. See "Packaging Client Applications" on page 3-13 for more about J2EE client applications.

CHAPTER

2 Developing WebLogic Server J2EE Applications

The following sections describe how to create different types of WebLogic Server J2EE applications (such as enterprise applications, Web applications, and Enterprise JavaBeans) and set up a development environment:

- Creating Web Applications: Main Steps
- Creating Enterprise JavaBeans: Main Steps
- Creating WebLogic Server Enterprise Applications: Main Steps
- Creating Resource Adapters: Main Steps
- Establishing a Development Environment
- Preparing to Compile
- Editing Deployment Descriptors

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.

Creating Web Applications: Main Steps

Creating a Web application requires creating HTML pages, JSPs, servlets, JSP taglibs, and two deployment descriptors, and then packaging everything into a `*.war` file. The `*.war` file is deployed on WebLogic Server as a Web application.

Here are the main steps for creating a Web application:

1. Create the HTML pages and 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* at http://e-docs.bea.com/wls/docs61/jsp/index.html.

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

   For detailed information about creating servlets, refer to *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html.

3. Compile the servlets into class files.

   For detailed information about compiling, refer to “Preparing to Compile” on page 2-17.

4. 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.

   You can create the `web.xml` and `weblogic.xml` deployment descriptors by hand, or you can use a Java-based utility included in WebLogic Server to automatically generate them. For more information on automatically generating these files, see “Automatically Generating Deployment Descriptors” on page 3-4.

   See *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html for detailed information on the elements in these deployment descriptors and instructions for creating them by hand.

5. Package the HTML pages, servlet class files, JSP files, web.xml, and weblogic.xml files into a Web archive (`*.war`) file.

   The first step in creating a `*.war` file is to create a Web application staging directory. JSP pages, HTML pages, and multimedia files referenced by the pages are saved in the top level of the staging directory. 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 about creating a `*.war` file, refer to "Packaging Web Applications" on page 3-6.

6. Auto-deploy the `*.war` file on WebLogic server for testing purposes.

   While you are testing the Web application you might need to edit the `web.xml` and `weblogic.xml` deployment descriptors; you can do this manually, or you can use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see "Editing Deployment Descriptors" on page 2-20.

   Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html for detailed information about auto-deploying components and applications.

7. 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. You use the Administration Console to deploy applications and components.

   Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html 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 up 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 for the interfaces and implementation into class files.

   For detailed information about compiling, refer to "Preparing to Compile" on page 2-17.

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 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.

   You can create the EJB deployment descriptors by hand, or you can use a Java-based utility included in WebLogic Server to automatically generate them. For more information on automatically generating these files, see "Automatically Generating Deployment Descriptors" on page 3-4.

   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* at http://e-docs.bea.com/wls/docs61/ejb/index.html.

4. Package the class files and deployment descriptors into a `*.jar` Java archive file

   The first step in creating a `*.jar` file is to 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 the stub and skeleton classes into the staging directory.

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` archive file, refer to "Packaging Enterprise JavaBeans" on page 3-8.

5. Auto-deploy the `*.jar` EJB archive file on WebLogic server for testing purposes.

   While you are testing the EJB you might need to edit the EJB deployment descriptors; you can do this manually, or you can use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see "Editing Deployment Descriptors" on page 2-20.

   Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html for detailed information about auto-deploying components and applications.

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. You use the Administration Console to deploy applications and components.

   Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html for detailed information about deploying components and applications.

Creating WebLogic Server Enterprise Applications: Main Steps

Creating a WebLogic Server enterprise application requires creating Web and EJB 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 and EJB components for your application.

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

For overview information about creating Web and EJB components, refer to "Creating Web Applications: Main Steps" on page 2-2 and "Creating Enterprise JavaBeans: Main Steps" on page 2-3.

For detailed information about creating the Java code that makes up the Web and EJB components, refer to *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html, *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html, and *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs61/jsp/index.html.

2. Create Web and EJB component 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` and `web.xml`. Additional deployment descriptors supplement the J2EE-specified descriptors with information required to deploy components in WebLogic Server.

   You can create the these deployment descriptors by hand, or you can use a Java-based utility included in WebLogic Server to automatically generate them. For more information on automatically generating these files, see "Automatically Generating Deployment Descriptors" on page 3-4.

   Refer to *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html for detailed information about writing Web component deployment descriptors by hand and to *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html for detailed information about writing EJB component deployment descriptors by hand.

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

   Component archives are JAR files containing all of the component files, including deployment descriptors. You package Web components into a `*.war` file and EJB components into an EJB `*.jar` file.

   Refer to "Packaging Web Applications" on page 3-6 and "Packaging Enterprise JavaBeans" on page 3-8 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.

   You can create the `application.xml` deployment descriptor by hand, or you can use a Java-based utility included in WebLogic Server to automatically generate it. For more information on automatically generating this file, see "Automatically Generating Deployment Descriptors" on page 3-4.

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

5. Package the enterprise application.

   Package the Web and EJB component archives along with the enterprise application deployment descriptor into an enterprise archive (`*.ear`) 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 Enterprise Application `*.ear` archive file, refer to "Packaging Enterprise Applications" on page 3-11.

6. Auto-deploy the `*.ear` enterprise application on WebLogic server for testing purposes.

   While you are testing the enterprise application you might need to edit the `application.xml` deployment descriptor; you can do this manually, or you can use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see "Editing Deployment Descriptors" on page 2-20.

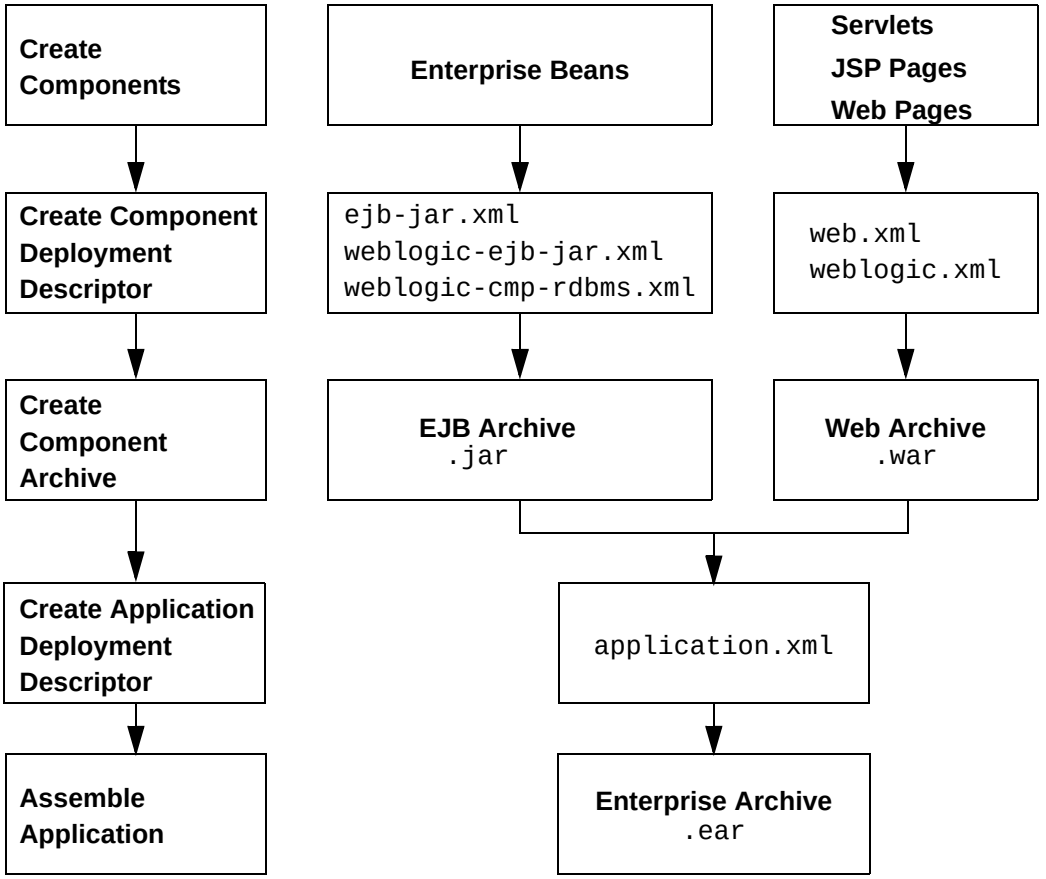
   Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html for detailed information about auto-deploying components and applications.

7. Deploy the `*.ear` file on WebLogic Server for production use. You use the Administration Console to deploy applications and components.

   Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html for detailed information about deploying components and applications.

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

Figure 2-1 Creating Enterprise 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 up into an `.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, Proposed Final Draft 2 (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 for the interfaces and implementation into class files.

3. 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. 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 like the following in the staging directory:

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

For detailed information about creating the resource adapter `.jar` archive file, refer to “Packaging Resource Adapters” on page 3-10.

4. 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 the WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/jconnector/index.html.

5. Create a resource adapter archive file (`.rar` file).
   a. The first step is to create an empty staging directory.
   b. Place the `.rar` file containing the resource adapter Java classes in the staging directory.
   c. Then, place the deployment descriptors in a subdirectory called `META-INF`.
   d. Next, create the resource adapter archive by executing a `jar` command like the following in the staging directory:

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

      For detailed information about creating the resource adapter archive file, refer to “Packaging Resource Adapters” on page 3-10.

6. Auto-deploy the `.rar` resource adapter archive file on WebLogic server for testing purposes.

   While you are testing the resource adapter you might need to edit the deployment descriptors; you can do this manually, or you can use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see “Editing Deployment Descriptors” on page 2-20.

Refer to *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html for detailed information about auto-deploying components and applications.

7. 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 *BEA WebLogic Server Administration Guide* at http://e-docs.bea.com/wls/docs61/adminguide/appman.html 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 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.jar
          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 the WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/jconnector/index.html 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.jar -C c:/stagedir
   ```

7. Deploy the resource adapter in WebLogic Server. For more information on deploying a resource adapter in WebLogic Server, see *Programming the WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/jconnector/index.html.

Establishing a Development Environment

To develop WebLogic Server applications, you need to assemble your software tools and set up an environment for creating, compiling, deploying, testing, and debugging your code. This section helps you start building your toolkit and setting up the compiler-related environment on your development computer.

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/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/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 “Editing Deployment Descriptors” on page 2-20.

Java Compiler

A Java compiler produces Java class files, containing portable byte code, from Java source. The compiler compiles the Java code you write for your applications, as well as the code generated by the WebLogic RMI, EJB, and JSP compilers.

Sun Microsystems Java 2, Standard Edition includes a Java compiler, `javac`. If you install the bundled JRE when you install WebLogic Server, the `javac` compiler is installed on your computer.

Other Java compilers are available for various platforms. You can use a different Java compiler for WebLogic Server application development as long as it produces standard Java `.class` files. Most Java compilers are many times faster than `javac`, and some are integrated nicely with an IDE.

Occasionally, a compiler generates optimized code that does not behave well in all Java Virtual Machines (JVMs). When you debug problems, try disabling optimizations, choosing a different set of optimizations, or compiling with `javac` to rule out your Java compiler as the cause. Always test your code in each target JVM before deploying.

Development WebLogic Server

Never deploy untested code on a WebLogic Server that is serving production applications. This means that you will need a development WebLogic Server in your environment. You can run a development WebLogic Server on the same computer you edit and compile on, or you can use one deployed somewhere 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 JMS require a supported JDBC driver for Oracle, Sybase, Informix, Microsoft SQL Server, IBM DB2, or Cloudscape. Refer to the Platform Support Web page at http://e-docs.bea.com/wls/certifications/certs_610/index.html 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. Connection pools are a collection of ready-to-use database connections. When a connection pool starts up, it creates a specified number of identical physical database connections. By establishing connections at start-up, the connection pool eliminates the overhead of creating a database connection for each application. BEA recommends that both client and server-side applications obtain connections from a connection pool through a Data Source on the JNDI tree. When finished with a connection, applications return the connection to the connection pool.

Multipools are multiplexers for basic connection pools. To the application they appear exactly as basic pools, but multipools allow you to establish a pool of connection pools, in which the connection attributes vary from connection pool to connection pool. All of the connections in a given connection pool are identical, but the connections in each connection pool in a multipool should vary in some significant way such that an expected failure of one pool will not invalidate another pool in the multipool. Usually these pools will be to different instances of the same database.

Multipools are only useful if there are multiple distinct database instances that can equally handle an application connection, and the application system takes care of synchronizing the databases when application work is distributed among the databases. In rare cases it may be valuable to have the pools to the same database instance, but as different users. This would be useful if the DBA disabled one user, leaving the other user viable.

By default, a clustered multipool provides high availability (DBMS failover). A multipool can be optionally configured to also provide load balancing.

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 SSL? 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 the *BEA WebLogic Developer Tools Resources* Web page at http://www.bea.com/products/weblogic/tools.shtml which provides developer tools information for products that support the BEA application servers.

To download some of these tools, see the *BEA WebLogic Server Downloads* Web page 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.

Preparing to Compile

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

- Have the 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

One way to set up your environment is to create a command file or shell script to set variables in your environment, which you can then pass to the compiler. The `setExamplesEnv.cmd` (Windows) and `setExamplesEnv.sh` (UNIX) files in the `config/examples` directory are examples of this technique.

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/java130` 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/java130/bin:$PATH; export PATH
```

To add the WebGain `sj` compiler to your path on Windows NT or Windows 2000, 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

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:

- If you are using the version of WebLogic Server 6.1 that is limited to J2EE 1.2 features (rather than the one that also includes J2EE 1.3 features), you *must* include the `j2ee12.jar` file in your CLASSPATH *before* you specify the `weblogic.jar` file. BEA recommends that you include the `j2ee12.jar` file in the beginning of your CLASSPATH.

  For more information on the version of J2EE (1.2 or 1.3) that your WebLogic Server instance implements, see "J2EE Platform" on page 1-3.

- The `lib/tools.jar` file in the JDK directory, or other standard Java classes required by the Java Development Kit you use.

- 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 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 `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`
: The directory where compiled client classes are written. These classes are usually standalone Java programs that connect to WebLogic Server. They do not have to be in the WebLogic Server CLASSPATH.

`SERVER_CLASSES`
: The directory where server-side classes are written. These classes include startup classes and other Java classes that must be in the WebLogic Server CLASSPATH when the server starts up. Application classes should usually not be compiled into this directory, because the classes in this directory cannot be redeployed without restarting WebLogic Server.

`EX_WEBAPP_CLASSES`
: The directory where classes used by the Web Application are written.

`APPLICATIONS`
: The `applications` directory for the examples domain. Unlike the others, 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 a make file, shell script, or command file to compile and package your components and applications. Set the variables in the build script so that you can rebuild components by typing a single command.

Editing Deployment Descriptors

You can edit the deployment descriptors of WebLogic applications and components using one of the following tools:

- 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`

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 provides you with the following two methods of editing the XML document:

- The hierarchical tree view allows structured, limited constrained editing, providing you 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 detailed information about using the BEA XML Editor, see its on-line help.

You can download BEA XML Editor from the *BEA dev2dev* at http://dev2dev.bea.com/resourcelibrary/utilitiestools/index.jsp.

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 you must explicitly click the Persist button. If you do not explicitly persist the changes to disk, the changes will be 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* at http://e-docs.bea.com/wls/docs61/ejb/index.html.

To edit the EJB deployment descriptors, follow these steps:

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 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 have made 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's memory.

Editing Web Application Deployment Descriptors

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

- `web.xml`
- `weblogic.xml`

See *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html for detailed information on the elements in the Web application deployment descriptors.

To edit the Web application deployment descriptors, follow these steps:

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 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, as described in the following list:

   - 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, 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 Web application 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 Web application 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 have made 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 if you want 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's memory.

Editing Resource Adapter Deployment Descriptors

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

- `ra.xml`
- `weblogic-ra.xml`

For detailed information about the elements in the resource adapter deployment descriptors, refer to *Programming the WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/jconnector/index.html.

To edit the resource adapter deployment descriptors, follow these steps:

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 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, as described in the following list:
   - 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, 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 resource adapter 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 resource adapter 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 have made 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 if you want 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's memory.

Editing Enterprise Application Deployment Descriptors

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

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

Note: The following procedure describes only how to edit the `application.xml` file; to edit the deployment descriptors in the components that make up the Enterprise application, see "Editing EJB Deployment Descriptors" on page 2-21, "Editing Web Application Deployment Descriptors" on page 2-23, or "Editing Resource Adapter Deployment Descriptors" on page 2-25.

To edit the Enterprise Application deployment descriptor, follow these steps:

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

8. Once you have made 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's memory.

CHAPTER

3 Packaging WebLogic Server J2EE Applications

The following sections describe how to package and deploy WebLogic Server J2EE applications:

- Packaging Overview
- Packaging Web Applications
- Packaging Enterprise JavaBeans
- Packaging Resource Adapters
- Packaging Enterprise Applications
- Packaging Client Applications
- Packaging J2EE Applications Using Apache Ant
- Packaging Client Applications

Packaging Overview

WebLogic Server J2EE applications are packaged in a standard way, defined by the 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 contains additional, WebLogic-specific deployment descriptors and, possibly, *container* classes generated with the WebLogic EJB, RMI, or JSP compilers.

JAR Files

A file created with the Java `jar` utility 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 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 (**t**ell). 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 required to deploy 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-INF/web.xml`
	WebLogic	`WEB-INF/weblogic.xml`

Table 3-1 J2EE and WebLogic Deployment Descriptors

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

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

You can create the deployment descriptors by hand, 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 3-4.

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 must extract the contents of the JAR file into a directory, add the required WebLogic-specific deployment descriptors and any generated container classes, and then create a new JAR file containing the old and new files.

Automatically Generating Deployment Descriptors

WebLogic Server includes a set of Java-based utilities that automatically generate the deployment descriptors for the following J2EE components or applications: Web applications, Enterprise JavaBeans (versions 1.1 and 2.0), and Enterprise Applications.

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.ant.taskdefs.ejb.DDInit`

  Creates the deployment descriptors for Enterprise JavaBeans 1.1.

- `weblogic.ant.taskdefs.ejb20.DDInit`

  Creates the deployment descriptors for Enterprise JavaBeans 2.0.

- `weblogic.ant.taskdefs.war.DDInit`

  Creates the deployment descriptors for Web applications.

- `weblogic.ant.taskdefs.ear.DDInit`

  Creates the deployment descriptors for Enterprise Applications.

Note: Although these utilities attempt 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" on page 2-20.

Each utility takes a single parameter: the root directory that contains the objects in the component or application for which you are generating deployment descriptors. The root directory is the one that contains the `WEB-INF` or `META-INF` subdirectories.

For example, assume that you have created a directory called `c:\stage` that contains the `WEB-INF` directory, 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.ant.taskdefs.war.DDInit c:\stage
```

The utility generates the `web.xml` and `weblogic.xml` deployment descriptors and places them in the `WEB-INF` directory.

Development Mode vs. Production Mode

You can run WebLogic Server in two different modes: development and production. You determine this mode by configuring the `STARTMODE` script variable, which is a variable you can modify in *`domain_name`*`\startWebLogic`. The `STARTMODE` variable allows you to toggle the start mode from production to development.

To enable development mode, configure the `STARTMODE` script variable as follows:

```
-Dweblogic.ProductionModeEnabled=false
```

To enable production mode, set the variable as follows:

```
-Dweblogic.ProductionModeEnabled=true
```

Note: The default setting is `false`.

For more information on starting WebLogic Server in development and production modes, refer to "Starting and Stopping WebLogic Servers."

When you specify development mode, you can use the auto-deploy feature of the `applications` directory. This means that you can copy new files into the `applications` directory of your Administration Server, located in the `config/`*`domain_name`* directory of the WebLogic Server installation (where *`domain_name`* is the name of a WebLogic Server domain). The aplications will be automatically deployed and updated.

In production mode, you must use the WebLogic Server Administration Console or the weblogic.Deploy tool to redeploy an application. Both deployment methods require a user name and password. This addresses security concerns around users who have write access to the file system and have the ability to deploy applications on the server.

Packaging Web Applications

Before you package your Web application, be sure you read and understand "Packaging Client Applications" on page 3-13 which describes how WebLogic server loads your application classes.

To stage and package a Web application:

1. Create a temporary staging directory. 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.

   Note: See "Classloader Overview" on page 3-21 to understand how the WebLogic Server class-loading mechanism affects EJB references from servlets within the same application.
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 *`BEA_HOME`*`\config\`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in the directory *`BEA_HOME`*`/config/`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.
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 3-4.

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

Note: See *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs61/webapp/index.html 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 the following:

   ```
   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.deploy` command-line utility.

Packaging Enterprise JavaBeans

You can stage one or more enterprise beans in a directory and package them in an EJB JAR file.

Before you package your EJBs, be sure you read and understand "Packaging Client Applications" on page 3-13 which describes how WebLogic server loads your EJB classes.

To stage and package an enterprise bean:

1. Create a temporary staging directory.
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 *`BEA_HOME`*`\config\`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in the directory *`BEA_HOME`*`/config/`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

5. Execute 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.ejb.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, use the following utility:

   ```
   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 3-4.

   Alternatively, you can create the EJB deployment descriptor files by hand. 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* at http://e-docs.bea.com/wls/docs61/ejb/index.html 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, you can 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 a WebLogic Server or package in an application JAR file.

   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*

at http://e-docs.bea.com/wls/docs61/ejb/index.html for help with the WebLogic EJB compiler.

Packaging Resource Adapters

You can stage one or more resource adapters in a directory and package them in a JAR file.

Before you package your resource adapters, be sure you read and understand "Packaging Client Applications" on page 3-13 which describes how WebLogic server loads classes.

To stage and package a resource adapter:

1. Create a temporary staging directory.
2. Compile or copy the resource adapter's Java classes into the staging directory.
3. Create a `META-INF` subdirectory in the staging directory.
4. 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:

   http://java.sun.com/dtd/connector_1_0.dtd

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

   Note: Refer to *Programming the WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs61/jconnector/index.html for information on the `weblogic-ra.xml` document type definition.

6. When all of the resource adapter classes and deployment descriptors are set up in the staging directory, you can create the resource adapter 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 a WebLogic Server or package in an application JAR file.

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 resource adapters.

Note: For instructions on creating a resource adapter and modifying an existing resource adapter for deployment to WebLogic Server, see "Creating Resource Adapters: Main Steps" on page 2-9 of Chapter 2, "Developing WebLogic Server J2EE Applications."

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 in another JAR file with an `.ear` extension.

The `META-INF` subdirectory in an `.ear` file contains an `application.xml` deployment descriptor, which 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`. No WebLogic-specific deployment descriptor is needed for an enterprise archive.

Here 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>
```

Before you package your enterprise application, be sure you read and understand "Packaging Client Applications" on page 3-13 which describes how WebLogic server loads your enterprise application classes.

To stage and package an Enterprise application:

1. Create a temporary staging directory.
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 *`BEA_HOME`*`\config\`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in the directory *`BEA_HOME`*`/config/`*`domain`*, where *`BEA_HOME`* is the directory in which WebLogic Server is installed and *`domain`* refers to the name of your domain.
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 3-4.

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

6. 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.deploy` command-line utility.

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.xml Deployment Descriptor Elements" in Appendix A, "application.xml 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.

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`

Compiling Java Source Files

Ant provides a `javac` task for compiling Java source files. The following example compiles all of the Java files in the current directory into a `classes` directory.

```
<target name=”compile”>
  <javac srcdir=”.” destdir=”classes”/>
</target>
```

Refer to Apache Ant online documentation for a full set of options relating to the `javac` task.

Running WebLogic Server Compilers

Running arbitrary Java programs from Ant can be accomplished by either writing custom Ant tasks or by simply executing the program using the `java` task. Tasks such as `ejbc` or `rmic` can be executed using the `java` task as shown below:

Listing 3-1 Running WebLogic Server Compilers

```
<java classname="weblogic.ejbc" fork="yes" failonerror="yes">
  <sysproperty key="weblogic.home" value="${WL_HOME}"/>
    <arg line="-compiler java ${dist}/std_ejb_basic_containerManaged.jar
     ${APPLICATIONS}/ejb_basic_containerManaged.jar"/>
  <classpath>
     <pathelement path="${CLASSPATH}"/>
  </classpath>
</java>
```

The above example uses the `fork` system call to create a Java process to run `ejbc`. The example supplies a `system` property to define `weblogic.home` and provide command line arguments using the `arg` tag. The classpath for the called Java process is specified using the `classpath` tag.

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 3-2 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 3-3 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
```

Resolving Class References Between Components

Your applications may use many different Java classes, including enterprise beans, servlets and JavaServer Pages, startup classes, 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.

Classloader Overview

A *classloader* is a Java class that locates and loads a requested class into the Java virtual machine (JVM). A classloader resolves references by searching for files in the directories or JAR files listed in its classpath. Most Java programs have a single classloader, the default system classloader created when the JVM starts up. WebLogic Server creates additional classloaders when it deploys applications because these classloaders can be destroyed in order to undeploy the application. This allows WebLogic Server to redeploy modified applications without having to restart the server.

Classloaders are hierarchical. When you start WebLogic Server, the Java system classloader is active and is the parent of all subsequent classloaders that WebLogic Server creates. A classloader always asks its parent for a class before it searches its own classpath, but a parent classloader does not consult its children. Because the search only proceeds upwards in the classloader hierarchy, this also means that a child classloader cannot locate classes on a sibling's classpath.

The search protocol also clarifies how duplicate classes are handled in Java. Classes located in the Java system classpath always have precedence over any class with the same name in a child classloader's classpath. Because of this, you should avoid placing application classes in the Java system classpath before you start WebLogic Server. The classloader created at startup time cannot be destroyed, so any classes it contains cannot be redeployed without restarting WebLogic Server.

About Application Classloaders

When WebLogic Server deploys an application, it creates two new classloaders: one for EJBs and one for Web applications. The EJB classloader is a child of the Java system classloader and the Web application classloader is a child of the EJB classloader. This allows classes in a Web application to locate EJB classes, but EJB classes cannot locate Web application classes. A positive side-effect of this classloader hierarchy is that it allows servlets and JSPs direct access to EJB implementation classes. WebLogic Server can bypass the intermediate RMI classes because the EJB client and implementation are in the same JVM.

If your application includes servlets and JSPs that use enterprise beans:

- 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 `.ejb` 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.

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 their corresponding archive file (for example, in the `.war`'s `/classes` directory for servlets or in the `.jar`'s `/classes` directory for EJBs).

Using PreferWebInfClasses in J2EE Applications

By default, the classloader for a web application follows the standard delegation model described in the Javasoft documentation. The servlet specification requires that a Web application obtain its class definition from the WAR file.

To support this requirement, BEA has included a switch that modifies the delegation model for a Web application so that the Web application's classloader looks for a class in the WAR file before asking its parent classloader for the class. This switch is called `PreferWebInfClasses` and is located on the `WebAppComponentMBean`. You can set this switch in the WebLogic Server console.

When you set `PreferWebInfClasses` to false (the default), the classloader for a Web application follows the standard delegation model. When set to true, it looks for class definitions in the WAR file before asking its parent for a class definition.

This switch satisfies the specification requirement. However, it leads to the possibility of having different versions of the same classes loaded in the Web application classloader than those versions existing in parent classloaders. This can lead to `ClassCastExceptions` if the developer is not careful to keep these two instances separate. For this reason, we have set the default for this setting to false, which means you use the standard delegation model.

Packaging Common Utilities and Third-Party Classes

If you create or acquire utility classes that you will use in more than one application, you must package them with each application. Alternatively, you could add them to the Java system classpath by editing the `java` command in the script that runs WebLogic Server. If you modify your utility classes and they are in the Java system classpath, however, you will have to restart WebLogic Server after you modify the utility classes.

Classes that WebLogic Server uses during startup must be in the Java system classpath. For example, JDBC drivers used for connection pools must be in the classpath when you start WebLogic Server. Again, if you need to modify classes in the Java system classpath, or modify the classpath itself, you will have to restart WebLogic Server after you modify the classes or the classpath.

Handling Interactions Between Startup Classes and Applications

Startup classes are classes you create that WebLogic Server executes at startup time. Startup classes are located by the Java system classpath, so you must put them in the system classpath before you start the server. Also, any classes they require must be included in the system classpath.

If a startup class uses application classes (such as EJB interfaces) you will also have to add those classes to the WebLogic Server startup classpath. Unfortunately, this means that you cannot modify those classes without restarting the server afterwards.

Startup classes that use application objects must wait for WebLogic Server to finish deploying the applications before the classes attempt to access the application objects. For example, if a startup class uses EJBs, you must include the home and remote interfaces in the system classpath, and you must ensure that the startup class does not create any EJB instances until WebLogic Server has finished deploying the EJB application.

The Pet Store application has a startup class that demonstrates one method a startup class can use to wait for applications to finish deploying. The `com.bea.estore.startup.StartBrowser` startup class displays the initial URL to

access the Pet Store application, and on Windows it also launches the browser with the URL. `StartBrowser` executes a `while` loop until applications have deployed and the server begins accepting connection requests.

Here is an excerpt from that class to show how this works:

```
while (loop) {
      try {
        socket = new Socket(host, new Integer(port).intValue());
        socket.close();

        //launch browser
        String[] cmdArray = new String[3];
        cmdArray[0] = "beaexec.exe";
        cmdArray[1] = "-target:browser";
        cmdArray[2] = "-command:\"http://"+host+":"+port+"\"";
        try {
          Process p = Runtime.getRuntime().exec(cmdArray);
          p.getInputStream().close();
          p.getOutputStream().close();
          p.getErrorStream().close();
        }
        catch (IOException ioe) {
        }
        loop = false;
      } catch (Exception e) {
        try {
          Thread.sleep(SLEEPTIME); // try every 500 ms
        } catch (InterruptedException ie) {}
        finally {
          try {
            socket.close();
          } catch (Exception se) {}
        }
      }
    }
```

If the system fails to create a socket, the class sleeps for 500 milliseconds before repeating the loop. If a startup class needs to create an EJB instance, it could use a similar technique by looping until the EJB create method succeeds.

CHAPTER

4 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
- Using Threads in WebLogic Server
- Using JavaMail with WebLogic Server Applications
- Programming Applications for WebLogic Server Clusters

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 use in client applications to write messages in a client-side log file.

This section describes how to use the `NonCatalogLogger` class. See the *BEA WebLogic Server Internationalization Guide* at http://e-docs.bea.com/wls/docs61/i18n/index.html for details on using the internationalization interface.

The log file name, location, and other properties can be administered in the Administration Console. Log messages written via the `NonCatalogLogger` class contain the following information.

Table 4-1 Log Message Format

Property	Description
Localized Timestamp	Date and time when message originated, including the year, month, day of month, hours, minutes and seconds.
millisecondsFromEpoch	The origination time of the message, in milliseconds since the epoch.
ServerName, MachineName, ThreadId, TransactionId	The origin of the message. TransactionId is present only for messages logged within the context of a transaction.
User Id	User on behalf of whom the system was executing when the error was reported.
Subsystem	Source of the message, for example EJB, JMS, or RMI. A user application supplies a Subsystem String in the `NonCatalogLogger` constructor.
Message Id	A unique six-digit identifier for the message. All message IDs through 499000 are reserved for WebLogic Server.

Table 4-1 Log Message Format

Property	Description	
Severity	One of the following severity values:	
	Debug	Should be output only when the server/application is configured in a debug mode. May contain detailed information about operations or the state of the server/application.
	Informational	Used to log normal operations for later examination.
	Warning	A suspicious operation, event, or configuration that does not affect the normal operation of the server/application.
	Error	A user level error. The system/application can handle the error with no interruption and with limited degradation in service.
	In addition to the above, some severity levels are reserved for WebLogic Server messages:	
	Notice	A warning message. A suspicious operation or configuration that does not affect the normal operation of the server.
	Critical	A system/service level error. The system is able to recover, perhaps with a momentary loss or permanent degradation of service.
	Alert	A particular service is in an unusable state. Other parts of the system continue to function. Automatic recovery is not possible and the immediate attention of the administrator is required to resolve the problem.
	Emergency	The server is in an unusable state. This is used to designate severe system failures or panics.
ExceptionName	If the message is logging an Exception, this field contains the name of the Exception.	
Message text	For WebLogic Server messages, this field contains the "short description" of the message defined in the system message catalog.	

To use `NonCatalogLogger`, import the `weblogic.logging.NonCatalogLogger` class and call the constructor with a subsystem String. Here is an example using the subsystem name "MyApp":

```
import weblogic.logging.NonCatalogLogger;
...
NonCatalogLogger mylogger = new NonCatalogLogger("MyApp");
```

`NonCatalogLogger` provides the methods `debug()`, `info()`, `warn()`, and `error()`, which write messages with Debug, Informational, Warning, and Error severities, respectively. Each method has two signatures, one that takes a String message argument, and another that takes a String message and a `java.lang.Throwable` argument. If you use the latter form, the log message includes a stack trace.

Here is an example of writing an informational message, without stack trace, to the log:

```
mylogger.info("MyApp initialized.");
```

If you are using `NonCatalogLogger` in a Java client, you specify the name of the log file on the `java` command line, using the `weblogic.log.FileName` Java system property. For example:

```
java -Dweblogic.log.FileName=myapp.log myapp
```

If you have special processing requirements for some log messages, you can add your own message handlers.Your message handler provides a filter to select the messages it is interested in processing. For each log message, the WebLogic Server logging infrastructure raises a JMX notification, which is delivered to the registered message handlers with filters that match the message.

See `weblogic.management.logging.WebLogicLogNotification` information about using this JMX feature.

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 you should construct your applications from components created using the standard J2EE APIs.

It is advisable to avoid application designs that require creating new threads in server-side components for several reasons:

- 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.

There are some situations where 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 decide you must use threads in your application code, your should 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 can help avoid performance issues and will also allow 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 IMAP- and 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, you should 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.

Table 4-2 Mail Session Properties 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.

Table 4-2 Mail Session Properties Field

Property	Description	Default
`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.

`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 and handle them.

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 Session Programming Requirements in 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 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 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 to understand the implications of binding clustered objects in the JNDI tree.

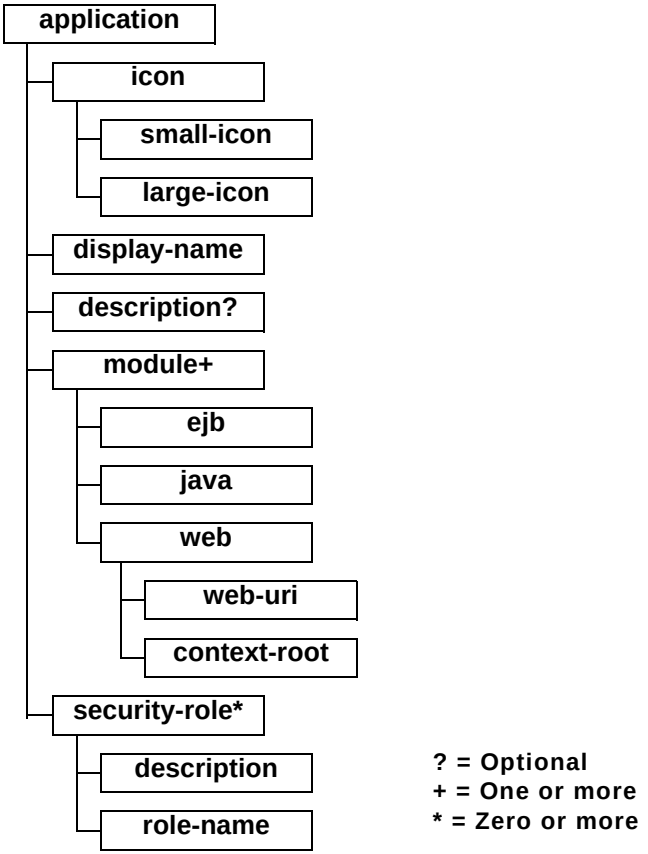
A application.xml Deployment Descriptor Elements

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 1.2//EN"
"http://java.sun.com/j2ee/dtds/application_1_2.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

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.

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 file. The `web` element contains a `web-uri` element and a `context-root` element.

web-uri

Defines the location of a Web module in the application file. This is the name of the `.war` file.

context-root

Required. 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

Required. 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 `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>
```

B Client Application Deployment Descriptor Elements

The following sections describe deployment descriptors for J2EE Client applications on WebLogic Server. Two deployment descriptors are required: a J2EE standard deployment descriptor, named `application-client.xml`, and a WebLogic-specific runtime deployment descriptor with a name derived from the client application JAR file.

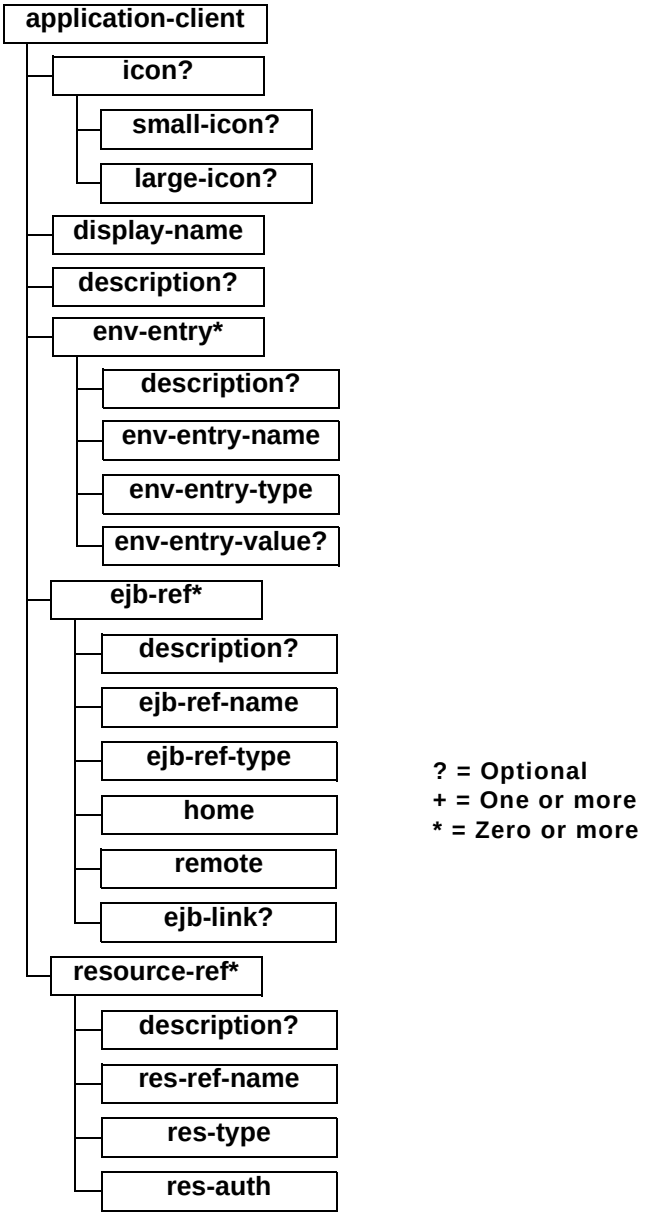
- application-client.xml Deployment Descriptor Elements
- WebLogic Run-time Client Application Deployment Descriptor

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

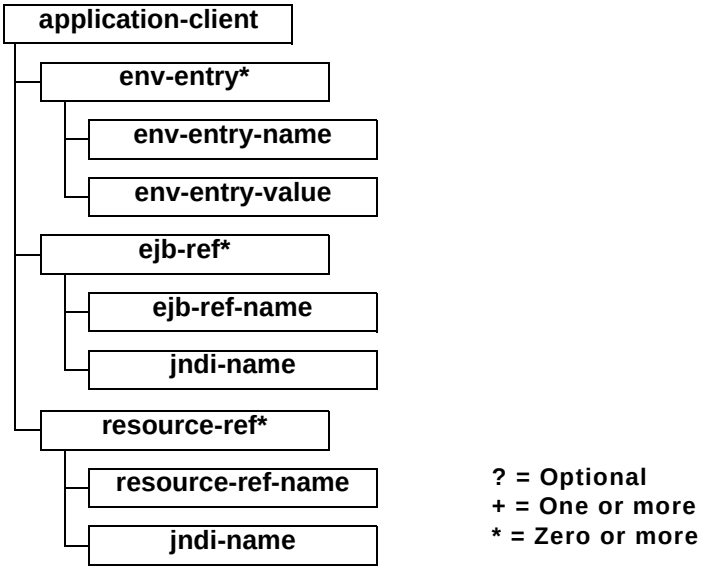
The `res-auth` element specifies whether the EJB code signs on programmatically to the resource manager, or whether the Container will sign on to the resource manager on behalf of the EJB. In the latter case, the Container uses information that is supplied by the Deployer. The res-auth element can have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