BEA WebLogic Server™

Developing WebLogic Server Applications

Release 7.0
Document Revised:December 15, 2005

Developing WebLogic Server Applications

Part Number	Document Revised	Software Version
N/A	August 20, 2002	BEA WebLogic Server Version 7.0

Contents

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, "WebLogic Server Application Classloading," provides an overview of Java classloaders, followed by details about WebLogic Server J2EE application classloading.
- Chapter 4, "WebLogic Server Application Packaging," describes how to bundle WebLogic Server components and applications in standard JAR files for distribution and deployment.
- Chapter 5, "WebLogic Server Deployment,"discusses WebLogic Server J2EE application deployment.
- Chapter 6, "Programming Topics," covers general WebLogic Server application programming issues, such as logging messages and using threads.
- Appendix A, "Application Deployment Descriptor Elements," is a reference for the standard J2EE Enterprise application deployment descriptor, `application.xml` and the WebLogic-specific application deployment descriptor `weblogic-application.xml`.
- Appendix B, "Client Application Deployment Descriptor Elements," is a reference for the standard J2EE Client application deployment descriptor,

`application-client.xml`, and the WebLogic-specific client application deployment descriptor.

Audience

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

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

- *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/index.html
- *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs70/servlet/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs70/jsp/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs70/webapp/index.html
- *Programming WebLogic JDBC* at http://e-docs.bea.com/wls/docs70/jdbc/index.html
- *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs70/webServices/index.html
- *Programming WebLogic J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs70/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. *Example*s: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n` *`name`* `-a` *`address`*
	     `[-p` *`portnumber`*`] [-t` *`timeout`*`] [-s` *`send`*`]`

`|`	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.Deployer [list|deploy|undeploy|update]`
	     `password {application} {source}`

`...`	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?

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

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

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

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

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

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

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

J2EE Platform

WebLogic Server implements Java 2 Platform, Enterprise Edition (J2EE) version 1.3 technologies (`http://java.sun.com/j2ee/sdk_1.3/index.html`). J2EE is the standard platform for developing multitier enterprise applications based on the Java programming language. The technologies that make up J2EE were developed collaboratively by Sun Microsystems and other software vendors, including BEA Systems.

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

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

Web Application Components

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

A Web application on WebLogic Server includes the following files:

- At least one servlet or JSP, along with any helper classes.
- A `web.xml` deployment descriptor, a J2EE standard XML document that describes the contents of a WAR file.
- A `weblogic.xml` deployment descriptor, an XML document containing WebLogic Server-specific elements for Web applications.

A Web application might also include HTML and XML pages with supporting files such as images and multimedia files.

Servlets

Servlets are Java classes that execute in WebLogic Server, accept a request from a client, process it, and optionally return a response to the client. A GenericServlet is protocol independent and can be used in J2EE applications to implement services accessed from other Java classes. An HttpServlet extends GenericServlet with support for the HTTP protocol. An HttpServlet is most often used to generate dynamic Web pages in response to Web browser requests.

JavaServer Pages

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

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

Web Application Directory Structure

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

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

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

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

See "Web Applications Directory Structure" in *Assembling and Configuring Web Applications* for detailed information on the Web application directory structure.

For More Information on Web Application Components

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

- *Programming WebLogic Server HTTP Servlets* at http://e-docs.bea.com/wls/docs70/servlet/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs70/jsp/index.html
- *Programming JSP Tag Extensions* at http://e-docs.bea.com/wls/docs70/taglib/index.html
- *Assembling and Configuring Web Applications* at http://e-docs.bea.com/wls/docs70/webapp/index.html

Enterprise JavaBean Components

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

EJB Overview

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

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

A message-driven bean is an enterprise bean that runs in the EJB container and handles asynchronous messages from a JMS Queue. When a message is received in the JMS Queue, the message-driven bean assigns an instance of itself from a pool to process the message. Message-driven beans are not associated with any client. They simply handle messages as they arrive. A JMS ServerSessionPool provides a similar capability but does not run in the EJB container.

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

EJB Interfaces

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

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

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

EJBs and WebLogic Server

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

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

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

For help creating and deploying EJBs, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/index.html.

Connector Component

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

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

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

To configure and deploy resource adapters, see *Programming the J2EE Connector Architecture* at http://e-docs.bea.com/wls/docs70/jconnector/index.html.

Enterprise Applications

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

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

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

WebLogic Web Services

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

A Web service consists of the following components:

- A Web service implementation hosted by a server on the Web.

WebLogic Web services are hosted by WebLogic Server. They are implemented using standard J2EE components (such as Enterprise Java Beans) and packaged as standard J2EE Enterprise Applications.

- A standardized way to transmit data and Web service invocation calls between the Web service and the user of the Web service.

  WebLogic Web Services use Simple Object Access Protocol (SOAP) 1.1 as the message format and HTTP as the connection protocol.

- A standard way to describe the Web service to clients so they can invoke it.

  WebLogic Web Services use Web Services Description Language (WSDL) 1.1, an XML-based specification, to describe themselves.

For information on designing, developing, and invoking WebLogic Web services, see *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs70/webServices/index.html.

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 through 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, 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`.

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 run a client application packaged to this specification. See "Packaging Client Applications" on page 4-23 for more about J2EE client applications.

Naming Conventions

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

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

CHAPTER

2 Developing WebLogic Server J2EE Applications

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

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

Creating Web Applications: Main Steps

Here are the main steps for creating a Web application:

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

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

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

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

3. Compile the servlets into class files.

   For detailed information about compiling, refer to "Compiling Java Code" on page 2-16.

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.

   Create the `web.xml` and `weblogic.xml` deployment descriptors manually or use a Java-based utility included in WebLogic Server to automatically generate them.

   See "Automatically Generating Deployment Descriptors" on page 4-5, and see *Assembling and Configuring Web Applications* for detailed information on the elements in these deployment descriptors and instructions for creating them manually.

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

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

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

For detailed information on creating WAR files, see "Packaging Web Applications" on page 4-16.

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 4-6.

   Refer to *BEA WebLogic Server Administration Guide* 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 Chapter 5, "WebLogic Server Deployment," for detailed information about deploying components and applications.

Creating Enterprise JavaBeans: Main Steps

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

Here are the main steps for creating an Enterprise JavaBean:

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

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

   For instructions on compiling, refer to "Compiling Java Code" on page 2-16.
3. Create the EJB-specific deployment descriptors:
   - `ejb-jar.xml` describes the EJB type and its deployment properties using a standard DTD from Sun Microsystems.
   - `weblogic-ejb-jar.xml` adds additional WebLogic Server-specific deployment information.
   - `weblogic-cmp-rdbms-jar.xml` maps a container-managed entity EJB to tables in a database. This file can must have a different name for each container-managed persistence (CMP) bean packaged in a JAR file. The name of the file is specified in the bean's entry in the `weblogic-ejb.jar` file.

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

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

Create and edit the XML deployment descriptors manually, or use 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 4-5.

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

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

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

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

   For detailed information about creating the EJB JAR file, refer to “Packaging Enterprise JavaBeans” on page 4-17.

5. Auto-deploy the EJB JAR 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 use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see “Editing Deployment Descriptors” on page 4-6.

   Refer to *BEA WebLogic Server Administration Guide* 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 Chapter 5, “WebLogic Server Deployment,” for detailed information about deploying components and applications.

Creating Resource Adapters: Main Steps

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

Creating a New Resource Adapter (RAR)

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

1. Write the Java code for the various classes required by resource adapter (ConnectionFactory, Connection, and so on) in accordance with the J2EE Connector Specification, Version 1.0, 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 using a standard compiler for the interfaces and implementation into class files.

   For instructions on compiling, refer to "Compiling Java Code" on page 2-16.

3. Create the resource connector-specific deployment descriptors:

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

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

   For detailed information about creating connector-specific deployment descriptors, refer to *Programming the WebLogic J2EE Connector Architecture*.

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

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

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

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

   For detailed information about creating the resource adapter RAR archive file, refer to "Packaging Resource Adapters" on page 4-20.

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

   During testing, you might need to edit the deployment descriptors. You can do this manually or use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see "Editing Deployment Descriptors" on page 4-6.

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

   Refer to Chapter 5, "WebLogic Server Deployment," for detailed information about deploying components and applications.

Modifying an Existing Resource Adapter (RAR)

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

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

```
mkdir c:/stagedir
```

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

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

3. Extract the contents of the resource adapter archive:

```
cd c:/stagedir

jar xf blackbox-notx.rar
```

   The staging directory should now contain the following:

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

   Execute these commands to see these files:

```
c:/stagedir> ls

        blackbox-notx.rar

        META-INF

c:/stagedir> ls META-INF

        Manifest.mf

        ra.xml
```

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

   Refer to *Programming the WebLogic J2EE Connector Architecture* for more information on the weblogic-ra.xml DTD.

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

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

c:/stagedir> ls META-INF

        Manifest.mf

        ra.xml
```

```
                weblogic-ra.xml
```

6. Create the resource adapter archive:

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

7. Deploy the resource adapter to WebLogic Server. There are several deployment tools. For detailed information about deploying components and applications, refer to Chapter 5, "WebLogic Server Deployment."

Creating WebLogic Server Enterprise Applications: Main Steps

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

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

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

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

   For overview information about creating Web, EJB, and Connector components, respectively refer to "Creating Web Applications: Main Steps" on page 2-2, "Creating Enterprise JavaBeans: Main Steps" on page 2-4, and "Creating Resource Adapters: Main Steps" on page 2-6.

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

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

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

Create these deployment descriptors manually, or you can use a Java-based utility included in WebLogic Server to generate them automatically. For more information on automatically generating these files, see "Automatically Generating Deployment Descriptors" on page 4-5.

For detailed information about manually creating, Web, EJB, and Connector deployment descriptors, refer to *Assembling and Configuring Web Applications*, *Programming WebLogic Enterprise JavaBeans*, and *Programming the WebLogic Server J2EE Connector Architecture*.

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

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

   Refer to "Packaging Web Applications" on page 4-16, "Packaging Enterprise JavaBeans" on page 4-17, and "Packaging Resource Adapters" on page 4-20 for detailed information for creating component archives.

4. Create the enterprise application deployment descriptor.

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

   Create the `application.xml` deployment descriptor manually, or use 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 4-5.

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

5. Package the enterprise application into an EAR file.

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

For detailed information about creating the EAR file, refer to "Packaging Enterprise Applications" on page 4-21.

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

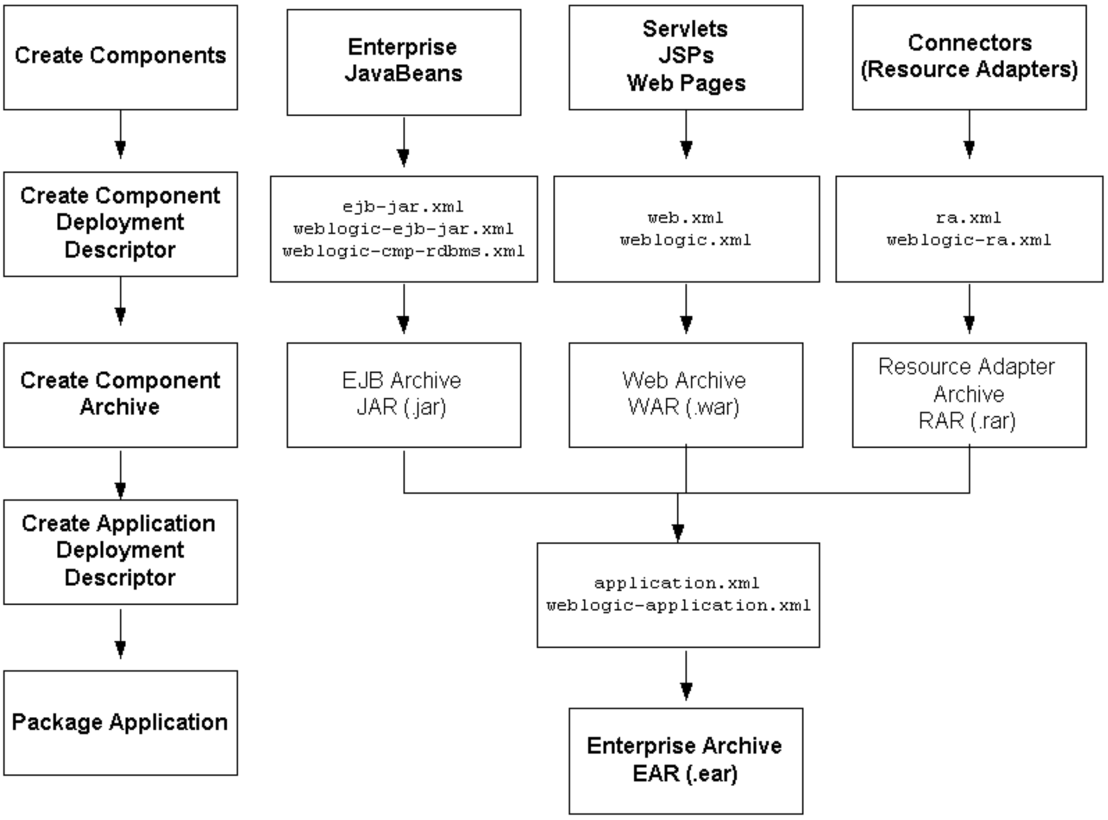
   While you are testing the enterprise application you might need to edit the `application.xml` deployment descriptor. Edit the file manually or use the deployment descriptor editor in the Administration Console. For detailed information on using the deployment descriptor editor, see "Editing Deployment Descriptors" on page 4-6.

7. For production purposes, use the Administration Console to deploy the EAR file on WebLogic Server.

   Refer to Chapter 5, "WebLogic Server Deployment," 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


Create Components
Create Component Deployment Descriptor
Create Component Archive
Create Application Deployment Descriptor
Package Application
Enterprise JavaBeans
Servlets JSPs Web Pages
Connectors (Resource Adapters)
ejb-jar.xml weblogic-ejb-jar.xml weblogic-cmp-rdbms.xml
web.xml weblogic.xml
ra.xml weblogic-ra.xml
EJB Archive JAR (.jar)
Web Archive WAR (.war)
Resource Adapter Archive RAR (.rar)
application.xml weblogic-application.xml
Enterprise Archive EAR (.ear)

Establishing a Development Environment

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

Software Tools

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

Source Code Editor or IDE

You need a text editor to edit Java source files, configuration files, HTML or XML pages, and JavaServer Pages. An editor that gracefully handles Windows and UNIX line-ending differences is preferred, but there are no other special requirements for your editor.

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

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

XML Editor

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

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

For detailed information about using these XML editors, see "Editing Deployment Descriptors" on page 4-6.

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 Java Runtime Environment (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. Instead, set up a development WebLogic Server instance on the same computer on which you edit and compile, or designate a WebLogic Server development location elsewhere on the network.

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

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

Database System and JDBC Driver

Nearly all WebLogic Server applications require a database system. You can use any DBMS that you can access with a standard JDBC driver, but services such as WebLogic Java Message Service (JMS) require a supported JDBC driver for Oracle, Sybase, Informix, Microsoft SQL Server, IBM DB2, or PointBase. Refer to *Platform Support* to find out about supported database systems and JDBC drivers.

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

Web Browser

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

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

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

Third-Party Software

You can use third-party software products, such as WebGain Studio, WebGain StructureBuilder, and BEA WebLogic Integration Kit for VisualAge for Java, to enhance your WebLogic Server development environment.

For more information, see *BEA WebLogic Developer Tools Resources*, which provides developer tools information for products that support the BEA application servers.

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

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

Compiling Java Code

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

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

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 `samples\server\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 Code

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

- The `lib/tools.jar` file in the JDK directory, or other standard Java classes required by the Java Development Kit you use.
- Classes for third party Java tools or services your programs import.
- Other application classes referenced by the programs you are compiling.

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

Setting Target Directories for Compiled Classes

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

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

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

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

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

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

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

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

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

If you do not use an IDE, consider writing 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.

CHAPTER

3 WebLogic Server Application Classloading

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

Java Classloader Overview

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

Java Classloader Hierarchy

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

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

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

Loading a Class

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

Note: Classloaders ask their parent classloader to load a class before attempting to load the class themselves. If needed, you can configure the classloader to check locally and then check the parent.

PreferWebInfClasses Element

The `WebAppComponentMBean` contains a `PreferWebInfClasses` element. By default, this element is set to `False`. Setting this element to `True` subverts the classloader delegation model and makes it very easy to obtain the same class loaded into both the Web application and system classloader, which in turn makes it very easy to obtain a `ClassCastException`.

Some users prefer to set this to `True` to override BEA implementations of various services (most commonly, XML processing classes such as `XERCES`). If you choose to set this element to `True`, be very careful not to mix instances of classes loaded from different classloaders.

Listing 3-1 PreferWebInfClasses Element

```
/**

* If true, classes located in the WEB-INF directory of a web-app
will be loaded in preference to classes loaded in the application
or system classloader.

* @default false

*/

boolean isPreferWebInfClasses();

void setPreferWebInfClasses(boolean b);
```

Changing Classes in a Running Program

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

When you deploy a new version of an application, a new application classloader is created. This scheme works as long as the application classes are being loaded by the new classloader. If a class is in the system classpath, it cannot be changed while the server is running.

Note: Java classloaders do not have any standard mechanism to undeploy or unload a set of classes; nor can they load new versions of classes. One way to get around this is to create an application-specific classloader as a child of the classpath classloader.

WebLogic Server Application Classloader Overview

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

Application Classloading

WebLogic Server classloading is centered on the concept of an application. An application is normally packaged in an Enterprise Archive (EAR) file containing application classes. Everything within an EAR file is considered part of the same application. Other applications include:

- An Enterprise JavaBean (EJB) JAR file
- A Web Application WAR file

Note: For information on Resource Adapter RAR files and classloading, see "About Resource Adapter Classes."

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

Make sure that no resource-adapter specific classes exist in your WebLogic Server system classpath. If you need to use resource adapter-specific classes with Web components (for example, an EJB or Web application), you must bundle these classes in the corresponding component's archive file (for example, the JAR file for EJBs or the WAR file for Web applications).

Every application receives its own classloader hierarchy; the parent of this hierarchy is the system classpath classloader. This isolates applications so that application A cannot see the classloaders or classes of application B. In classloaders, no sibling or friend concepts exist. Application classloaders can only see their parent classloader, the system classpath classloader. This allows WebLogic Server to host multiple isolated applications within the same JVM.

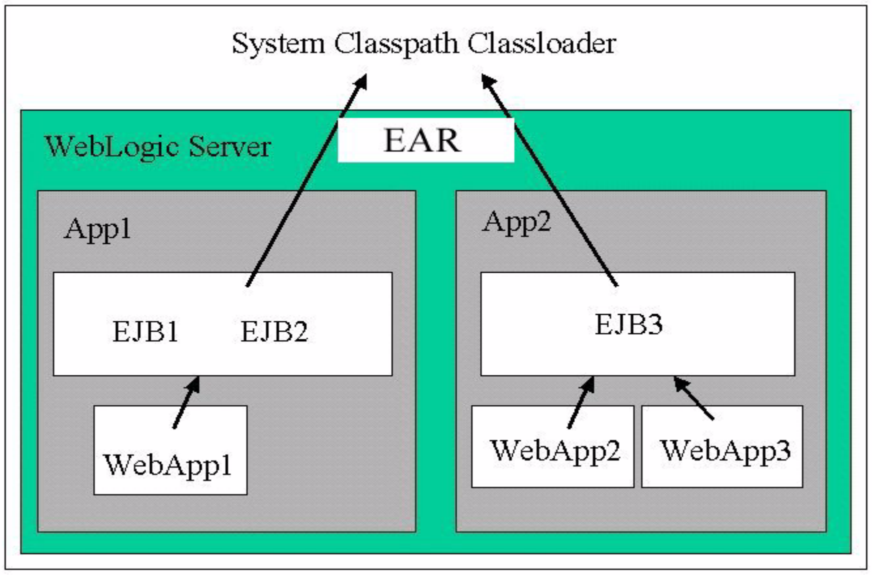
Application Classloader Hierarchy

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

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

The following graphic illustrates this WebLogic Server application classloading concept:

Figure 3-1 WebLogic Server Classloading



If your application includes servlets and JSPs that use EJBs:

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

Although you could deploy the WAR and JAR files separately, deploying them together in an EAR file produces a classloader arrangement that allows the servlets and JSPs to find the EJB classes. If you deploy the WAR and JAR files separately, WebLogic Server creates sibling classloaders for them. This means that you must include the EJB home and remote interfaces in the WAR file, and WebLogic Server must use the RMI stub and skeleton classes for EJB calls, just as it does when EJB clients and implementation classes are in different JVMs. This concept is discussed in more detail in the next section "Application Classloading and Pass by Value or Reference" on page 3-7.

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

Application Classloading and Pass by Value or Reference

Modern programming languages use two common parameter passing models: pass by value and pass by reference. With pass by value, parameters and return values are copied for each method call. With pass by reference, a pointer (or reference) to the actual object is passed to the method. Pass by reference improves performance because it avoids copying objects, but it also allows a method to modify the state of a passed parameter.

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

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

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

The following is a list of pass-by-value and pass-by-reference parameters called by EJB versions 2.0 and 1.1.

Table 3-1 Parameters Called by EJB Version 2.0

Packaging	Called Interface	Default Call Type	Effect of enable-call-by-reference
Caller WebApp/EJB are in an EAR file	Local	Pass-by-Reference	Pass-by-Value if False
	Remote	Pass-by-Value	Pass-by-Reference if True
Caller EJB and called EJB are in the same JAR file, not the same EAR file	Local	Pass-by-Reference	Pass-by-Value if False
	Remote	Pass-by-Value	Pass-by-Reference if True
Caller WebApp/EJB and called EJB are in separate deployment module (EAR/JAR)	Local	Pass-by-Value	N/A
	Remote	Pass-by-Value	N/A

Table 3-2 Parameters Called by EJB Version 1.1

Packaging	Called Interface	Default Call Type	Effect of enable-call-by-reference
Caller WebApp/EJB are in an EAR file	Remote	Pass-by-Reference	Pass-by-Value if False
Caller EJB and called EJB are in the same JAR file, not the same EAR file	Remote	Pass-by-Reference	Pass-by-Value if False
Caller WebApp/EJB and called EJB are in separate deployment module (EAR/JAR)	Remote	Pass-by-Value	N/A

Resolving Class References Between Components and Applications

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

About Resource Adapter Classes

Make sure that no resource-adapter specific classes exist in your WebLogic Server system classpath. If you need to use resource adapter-specific classes with Web components (for example, an EJB or Web application), you must bundle these classes in the corresponding component's archive file (for example, the JAR file for EJBs or the WAR file for Web applications).

Packaging Shared Utility Classes

Applications usually have shared utility classes. If you create or acquire utility classes that you will use in more than one application, you must package them with each application as separate JAR files. The JAR files should be self contained and not have any references to the classes in the EJB or Web components. Common types of shared utility classes are data transfer objects or JavaBeans, which are passed between the Web tier and EJB tier.

Alternatively, you can add shared utility classes 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.

Manifest Class-Path

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

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

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

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

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

```
/<directory>/x.war
```

```
/<directory>/y.jars
```

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

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

CHAPTER

4 WebLogic Server Application Packaging

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

Packaging Overview

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

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

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

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

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

JAR Files

A file created with the Java `jar` tool bundles the files in a directory into a single Java ARchive (JAR) file, maintaining the directory structure. The Java classloader can search for Java class files (and other file types) in a JAR file the same way that it searches a directory in its classpath. Because the classloader can search a directory or a JAR file, you can deploy J2EE components on WebLogic Server in either an "exploded" directory or a JAR file.

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

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

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

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

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

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

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

XML Deployment Descriptors

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

Table 4-1 lists the types of components and applications and their J2EE-standard and WebLogic-specific deployment descriptors.

Table 4-1 J2EE and WebLogic Deployment Descriptors

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

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

You can create the deployment descriptors manually, or you can use WebLogic-specific Java-based utilities to automatically generate them for you. For more information about generating deployment descriptors, see "Automatically Generating Deployment Descriptors" on page 4-5.

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

Automatically Generating Deployment Descriptors

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

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

WebLogic Server includes the following utilities:

- `weblogic.marathon.ddinit.WebInit`

  Creates the deployment descriptors for Web Applications.

- `weblogic.marathon.ddinit.EJBInit`

  Creates the deployment descriptors for Enterprise JavaBeans 2.0.If `ejb-jar.xml` exists, `DDInit` uses its deployment information to generate `weblogic-ejb-jar.xml`.

Limitations of DDInit

`DDInit` attempts to create deployment descriptor files that are complete and accurate for your component or application, but must guess at the value of many of the required elements. If such a guess is wrong, WebLogic Server will return an error when you deploy the component or application. If this happens, 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."

Relations among entity beans are a particular problem for DDInit, because it can only be sure of one-to-one bi-directional relations. If a relation has a "many" side to it, DDInit will guess at the nature of the relation.

Example

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

```
java weblogic.marathon.ddinit.WebInit c:\stage
```

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

Editing Deployment Descriptors

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

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

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

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

- `ra.xml`
- `weblogic-ra.xml`
- `application.xml`
- `weblogic-application.xml`
- `application-client.xml`
- `client-application.runtime.xml`

Using the BEA XML Editor

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

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

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

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

About EJBGen

EJBGen is an Enterprise JavaBeans 2.0 code generator or command-line tool that uses Javadoc markup to generate EJB deployment descriptor files. You annotate your Bean class file with javadoc tags and then use EJBGen to generate the Remote and Home classes and the deployment descriptor files for an EJB application, reducing to one the number of EJB files you need to edit and maintain.

For more information about EJBGen, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs70/ejb/EJB_utilities.html.

Using the Administration Console Deployment Descriptor Editor

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

When you use the editor, you can either update the in-memory deployment descriptor only, or update both the in-memory and disk files. When you click the Apply button after updating a particular element, or the Create button to create a new element, only the deployment descriptor in WebLogic Server's memory is updated; the change has not yet been written to disk. To do this, click the Persist button. If you do not explicitly persist the changes to disk, the changes are lost when you stop and restart WebLogic Server.

Editing EJB Deployment Descriptors

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

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

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

To edit the EJB deployment descriptors:

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

   `http://host:port/console`

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

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

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

5. To edit, delete, or add elements in the EJB deployment descriptors, click to expand the node in the left pane that corresponds to the deployment descriptor file you want to edit, as described in the following list:
   - The EJB JAR node contains the elements of the `ejb-jar.xml` deployment descriptor.
   - The WebLogic EJB Jar node contains the elements of the `weblogic-ejb-jar.xml` deployment descriptor.
   - The container-managed persistence (CMP) node contains the elements of the `weblogic-cmp-rdbms-jar.xml` deployment descriptor.
6. To edit an existing element in one of the EJB deployment descriptors, follow these steps:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the element you want to edit.
   b. Click the element. A form appears in the right pane that lists either its attributes or sub-elements.
   c. Edit the text in the form in the right pane.
   d. Click Apply.
7. To add a new element to one of the EJB deployment descriptors, follow these steps:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to create.
   b. Right-click the element and chose Configure a New *Element* from the drop-down menu.
   c. Enter the element information in the form that appears in the right pane.
   d. Click Create.
8. To delete an existing element from one of the EJB deployment descriptors, follow these steps:

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

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

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

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

10. Click Validate if you want to ensure that the entries in the EJB deployment descriptors are valid.

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

Editing Web Application Deployment Descriptors

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

See *Assembling and Configuring Web Applications* for detailed information on the elements in the Web application deployment descriptors.

To edit the Web application deployment descriptors:

1. Invoke the Administration Console in your browser:

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

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

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

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

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

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

5. To edit, delete, or add elements in the Web application deployment descriptors, click to expand the node in the left pane that corresponds to the deployment descriptor file you want to edit:
   - The Web App Descriptor node contains the elements of the `web.xml` deployment descriptor.
   - The WebApp Ext node contains the elements of the `weblogic.xml` deployment descriptor.
6. To edit an existing element in one of the Web application deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the element you want to edit.
   b. Click the element. A form appears in the right pane that lists either its attributes or sub-elements.
   c. Edit the text in the form in the right pane.
   d. Click Apply.
7. To add a new element to one of the Web application deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to create.
   b. Right-click the element and chose Configure a New *Element* from the drop-down menu.
   c. Enter the element information in the form that appears in the right pane.
   d. Click Create.
8. To delete an existing element from one of the Web application deployment descriptors:
   a. Navigate the tree in the left pane, clicking on parent elements until you find the name of the element you want to delete.
   b. Right-click the element and choose Delete *Element* from the drop-down menu.
   c. Click Yes to confirm that you want to delete the element.

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

10. Click Validate to ensure that the entries in the Web application deployment descriptors are valid.

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

Editing Resource Adapter Deployment Descriptors

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

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

To edit the resource adapter deployment descriptors:

1. Invoke the Administration Console in your browser:

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

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

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

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

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

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

5. To edit, delete, or add elements in the resource adapter deployment descriptors, click to expand the node in the left pane that corresponds to the deployment descriptor file you want to edit:

- The RA node contains the elements of the `ra.xml` deployment descriptor.
- The WebLogic RA node contains the elements of the `weblogic-ra.xml` deployment descriptor.

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

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

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

Editing Enterprise Application Deployment Descriptors

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

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

Note: The following procedure describes only how to edit the `application.xml` and `weblogic-application.xml` files; to edit the deployment descriptors in the components that make up the Enterprise application, see "Editing EJB Deployment Descriptors" on page 4-8, "Editing Web Application Deployment Descriptors" on page 4-10, or "Editing Resource Adapter Deployment Descriptors" on page 4-12.

To edit the Enterprise Application deployment descriptor:

1. Invoke the Administration Console in your browser:

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

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

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

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

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

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

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

Packaging Web Applications

If your Web application is accessed by a programmatic Java client, see "Packaging Client Applications" on page 4-23, which describes how WebLogic server loads your application classes.

To stage and package a Web application:

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

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

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

```
   java weblogic.marathon.ddinit.WebInit 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 4-5.

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

Note: See *Assembling and Configuring Web Applications* for detailed descriptions of the elements of the `web.xml` and `weblogic.xml` files.

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

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

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

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

Packaging Enterprise JavaBeans

You can stage one or more Enterprise JavaBeans (EJBs) in a directory and package them in an EJB JAR file. If your EJB is accessed by a programmatic Java client, see "Packaging Client Applications" on page 4-23 which describes how WebLogic Server loads your EJB classes.

Staging and Packaging EJBs

To stage and package an Enterprise JavaBean (EJB):

1. Create a temporary staging directory anywhere on your hard drive (for example, `c:\stagedir`).

2. Compile or copy the bean's Java classes into the staging directory.

3. Create a `META-INF` subdirectory in the staging directory.

4. Set up your shell environment.

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

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

5. 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.marathon.ddinit.EJBInit staging-dir
   ```

   where *`staging-dir`* refers to the staging directory. This utility generates deployment descriptors for EJB 2.0.

   For more information on the Java-based `DDInit` utility for generating deployment descriptors, see "Automatically Generating Deployment Descriptors" on page 4-5.

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

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

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

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

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

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

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

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

Using ejb-client.jar

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

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

Packaging Resource Adapters

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

To stage and package a resource adapter:

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

   Note: Refer to the following Sun Microsystems documentation for information on the `ra.xml` document type definition at: `http://java.sun.com/dtd/connector_1_0.dtd`
6. Create a `weblogic-ra.xml` deployment descriptor in the `META-INF` subdirectory and add entries for the resource adapter.

   Note: Refer to *Programming WebLogic J2EE Connectors* for information on the `weblogic-ra.xml` document type definition.
7. When the resource adapter classes and deployment descriptors are set up in the staging directory, you can create the RAR with a JAR command such as:

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

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

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

Packaging Enterprise Applications

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

Enterprise Applications Deployment Descriptor Files

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

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

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

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

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

<application>
  <display-name>estore</display-name>
  <description>Application description</description>
  <module>
    <web>
      <web-uri>petStore.war</web-uri>
      <context-root>estore</context-root>
    </web>
  </module>
  <module>
    <ejb>petStore_EJB.jar</ejb>
  </module>
  <security-role>
    <description>the gold customer role</description>
    <role-name>gold_customer</role-name>
  </security-role>
  <security-role>
    <description>the customer role</description>
    <role-name>customer</role-name>
  </security-role>
</application>
```

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

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

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

For information about application scoping, see Application Scoped JDBC Connection Pools at http://e-docs.bea.com/wls/docs70/jdbc/programming.html#1050534, and XML Application Scoping at http://e-docs.bea.com/wls/docs70/xml/xml_appscop.html.

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

Packaging Enterprise Applications: Main Steps

If your enterprise application is accessed by a programmatic Java client, see "Packaging Client Applications" on page 4-23, which describes how WebLogic Server loads your enterprise application classes.

To stage and package an Enterprise application:

1. Create a temporary staging directory anywhere on your hard drive.
2. Copy the Web archives (WAR files) and EJB archives (JAR files) into the staging directory.

3. Create a `META-INF` subdirectory in the staging directory.
4. Set up your shell environment.

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

   On UNIX, execute the `setenv.sh` command, located in the directory *`server`*`/bin/`*`setenv.sh`*, where *`server`* is the directory in which WebLogic Server is installed.
5. Create the `application.xml` deployment descriptor file that describes the enterprise application in the `META-INF` directory. See Appendix A, "Application Deployment Descriptor Elements," for detailed information about the elements in this file.
6. Optionally create the `weblogic-application.xml` file manually in the `META-INF` directory, as described in Appendix A, "Application Deployment Descriptor Elements."
7. Create the Enterprise Archive (EAR file) for the application, using a `jar` command such as:

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

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

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

Packaging Client Applications

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

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

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

Executing a Client Application in an EAR File

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

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

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

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

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

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

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

For example:

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

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

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

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

For example

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

Special Considerations for Deploying J2EE Client Applications

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

- Name the WebLogic Server client deployment file using the suffix `.runtime.xml`.
- The `weblogic.ClientDeployer` class is responsible for generating and adding a `client.properties` file to the client JAR file. A separate program, `weblogic.j2eeclient.Main`, creates a local client JNDI context and runs the client from the entry point named in the client manifest file.

  Note: To run the J2EE client application using `weblogic.ClientDeployer`, you need the `weblogic.j2eeclient.Main` class (located in the `weblogic.jar` file).
- If a resource mentioned by the `application-client.xml` file is one of the following types, the `weblogic.j2eeclient.Main` class attempts to bind it from the global JNDI tree on the server to `java:comp/env/`:

  `ejb-ref`

  `javax.jms.QueueConnectionFactory`

```
javax.jms.TopicConnectionFactory

javax.mail.Session

javax.sql.DataSource
```

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

Note: For more information on deploying, refer to Chapter 5, "WebLogic Server Deployment."

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 4-1 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 4-2 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 4-3 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

CHAPTER

5 WebLogic Server Deployment

This release of WebLogic Server introduces a new deployment protocol, two-phase deployment, which helps prevent inconsistent deployment states across servers, especially clustered servers. See Two-Phase Deployment on page 2.

You can deploy an application using the WebLogic Server Administration Console, `weblogic.Deployer` utility, WebLogic Builder, or auto-deployment. These deployment tools are discussed in Deployment Tools and Procedures.

The following sections discuss WebLogic Server deployment:

Two-Phase Deployment

The new two-phase deployment protocol helps to maintain domain consistency. In previous versions of WebLogic Server, when you deployed an application, the administration server sent a copy of the application file(s) to all the targeted servers, which then loaded the application. If deployment to any of those servers failed or partially failed, the entire deployment's state across its target servers became inconsistent.

In the current release of WebLogic Server, deployment first prepares the application across all target servers and then activates the application in a separate phase. If a deployment of an application fails in either the preparation or activation phase, then the cluster deployment is failed.

For information about using the earlier WebLogic Server deployment protocol, see Using WebLogic Server 6.x Deployment Protocol.

The new deployment protocol supports the following new features for deployed applications:

- **Consistent deployment states for clusters**. If an application targeted to a cluster fails on any of the cluster members in the prepare phase and then in the activate phase, the application is not activated on any of the cluster members. This helps to ensure that the cluster is kept homogeneous.
- **Application ordering**. At server startup, you set the order of application activations. See Deployment Order for Resources and Applications.
- **Application-scoped configuration**. Certain resources can be configured and scoped for an application. These include connection pools, security realms and XML related resources. See Overview of Application Scoping.
- **Improved redeployment**. You do not need to undeploy before redeploying. See Updating Applications with the Administration Console, Undeploying and Redeploying Archived Applications, and Redeploying Applications in Exploded Format.

  Note: An application becomes unavailable to clients during redeployment. For this reason, redeployment is not recommended for use in a production environment.

- **Improved API**. A simple API separates configuration from the actual deployment operations. When a deployment is requested, this API creates the necessary configuration (MBeans) for you. Also, the deployment operations are not on the MBeans themselves, so you can change the configuration (such as the target lists) without affecting the deployed application, until a deployment request is initiated. See Deployment Management API, and see also the API documentation at `weblogic.management.deploy.`
- **Deployment status**. It is now easier to track the progress of a deployment especially when it has multiple targets. See Example Uses of the weblogic.Deployer Utility, and WebLogic Administration Console help on Tasks.

Restarting Admin Server

Stopping and restarting your Admin Server cancels pending deployment requests. If your are deploying to a cluster and one of the targeted servers in the cluster is down, an Admin Server restart will prevent the targeted server from receiving the deployment request when the targeted server comes back up.

Prepare Phase and Activate Phase

The two-phase model makes inconsistent deployment states in clusters less likely by confirming the success of the prepare phase before deploying the application on any targeted servers. A deployment that fails during the prepare phase will not enter the activation phase.

Prepare Phase

The prepare phase of deployment, the first phase, distributes or copies files and prepares the application and its components for activation, validating them and performing error checks on them. The purpose of the prepare phase is to ensure that the application and its components are in a state in which they can be reliably deployed.

Activate Phase

The second phase, the activate phase, is the actual deployment, or activation, of the application and its component with the relevant server subsystem. After the activate phase, the application is made available to clients.

Deployment Order for Resources and Applications

By default, WebLogic Server deploys server-level resources (JDBC followed by JMS) before deploying applications and standalone modules, followed by startup classes. The order of startup class execution is configurable, as described in "Ordering Startup Class Execution and Deployment" on page 5-5.

Setting the Order of Applications

Applications are deployed in this order: connectors, then EJBs, then Web Applications. WebLogic Server 7.0 allows you to select the load order for applications. See the `ApplicationMBean LoadOrder` attribute in Application.

Ordering Components Within an Application

If the application is an EAR, the individual components are loaded in the order in which they are declared in the `application.xml` deployment descriptor. See Editing Enterprise Application Deployment Descriptors.

Ordering Startup Class Execution and Deployment

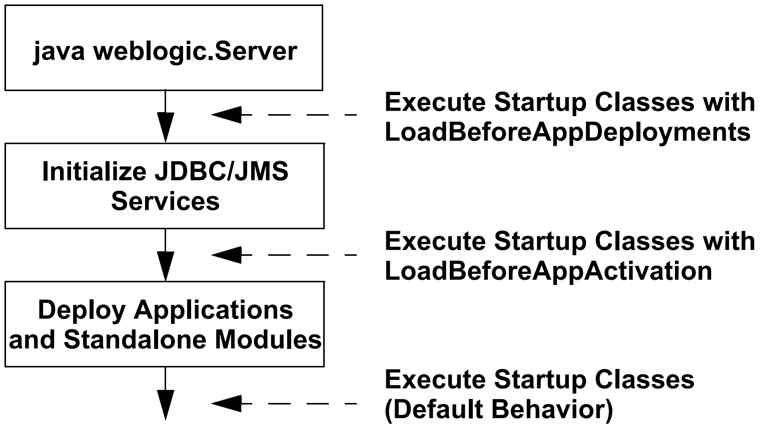
By default WebLogic Server startup classes are run after the server initializes JMS and JDBC services, and after applications and standalone modules have been deployed.

If you want to perform startup tasks after JMS and JDBC services are available, but before applications and modules have been activated, you can select the Run Before Application Deployments option in the Administration Console (or set the `StartupClassMBean`'s `LoadBeforeAppActivation` attribute to "true").

If you want to perform startup tasks before JMS and JDBC services are available, you can select the Run Before Application Activations option in the Administration Console (or set the `StartupClassMBean`'s `LoadBeforeAppDeployments` attribute to "true").

The following figure summarizes the time at which WebLogic Server executes startup classes.

Figure 5-1 Startup Class Execution


java weblogic.Server
Execute Startup Classes with LoadBeforeAppDeployments
Initialize JDBC/JMS Services
Execute Startup Classes with LoadBeforeAppActivation
Deploy Applications and Standalone Modules
Execute Startup Classes (Default Behavior)

BEA provides the `examples.jms.startup` API code example which demonstrates how to establish a JMS message consumer from a WebLogic startup class. See the full Javadocs for `StartupClassMBean` for more information.

Note: WebLogic Server 7.0 optionally installs API code examples in `WL_HOME`\samples\server\src\examples, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them.

Application Staging

In this release of WebLogic Server, the staging mode controls whether or not, where, and by whom application files are copied for deployment. Staging means the copying of application files to the locations from which they will be deployed. Each server has a staging mode. Once an application is deployed, its staging mode cannot be changed. See Best Practices for Application Deployment for help on when to use staging modes.

Staging Modes

Available staging modes are:

- `nostage`: does not copy application files to another location.

  A server in nostage mode will run applications deployed to it directly from their source directories. In this mode, the web application container detects changes to JSPs and servlets.

- `stage`: copies application files to server targeted in deployment

  The stage mode means that the Administration Server copies source files to the staging directory on target servers when you perform a deployment operation. The target servers then initialize and run the application from this directory.

- `external_stage`: the user, and not WebLogic Server, ensures that application files are copied to the server's staging directory before deployment.

  The deployment should be copied to a directory with the same name as the application name under each target server's staging directory.

The external stage mode means that the application will be run from a staging directory, to which an external entity is expected to distribute the files. This mode is useful in environments that are managed by third-party tools.

Note: In order to use either `nostage` or `external_stage` modes, the files to deploy must be accessible to the Administration Server machine. You can either copy the files to the Administration Server machine or place them on a shared directory that is available from the Administration Server machine.

Staging mode defaults are:

- for managed servers, staging mode is `stage` by default, meaning that the default staging behavior is to copy the application files to their targeted managed servers
- for administration servers, staging mode is `nostage` by default, meaning that the default staging behavior is to deploy from the source location provided

The following table describes how staging attribute and path settings affect an application's deployment:

Table 5-1 Deployment Staging Modes

Staging Mode	Staging Directory	Administration Server Deployment Behavior	Managed Server Deployment Behavior
stage	Absolute, Relative	Application files are copied to a directory named after the application deployment in the server staging directory, and activated from there (e.g., `server/stage/myapp/app.ear`). If the staging path is relative, the path is evaluated relative to the server's root directory.	Same as for Administration Server.
nostage	Absolute, Relative	Staging path is ignored, and no files are copied. Files are deployed from their location on the Administration Server, or from a shared directly accessible by the Administration Server.	Same as for Administration Server. (Source files must be accessible from the Managed Server machine in order to deploy.)

Table 5-1 Deployment Staging Modes

Staging Mode	Staging Directory	Administration Server Deployment Behavior	Managed Server Deployment Behavior
external_stage	Absolute, Relative	No files are copied, but deployment files are assumed to reside in a subdirectory named after the deployment in the server staging directory. and are loaded from there. For example, if you deploy an application using the deployment name "myextapp" and the server staging directory is `.\myserver\stage`, you must ensure that the deployment files are available in `.\myserver\stage\myextapp` before deploying. If the staging path is relative, the path is evaluated relative to the server's root directory.	Same as for Administration Server. (You must ensure deployment files are copied to the correct subdirectory of the Managed Server's staging directory.)

Configuring Staging Modes and Directories

By default, when you deploy an application to a managed server, it is staged to the target server staging area (`ServerMBean.StagingDirectoryName`) and deployed from there. You can disable staging using the `ServerMBean.StagingMode` attribute or the `ApplicationMBean.StagingMode` attribute. The `ServerMBean.StagingMode` attribute applies to all applications deployed to that server. It can be overridden by `ApplicationMBean.StagingMode`.

See Best Practices for Application Deployment for help on when to use staging modes.

For Javadoc on the attributes mentioned here, see Javadocs for WebLogic Classes.

Staging Scenarios

You can configure the system to perform some common tasks described in the sections that follow.

Deploy Application from its Source Location

Configure the `StagingMethod` attribute on the application or on a specific server to be set to `nostage`. If you configure this on the application, you must do so when the application is configured in the server using the `weblogic.Deployer` tool.

This mode is useful for incremental development on a single server, and is also handy in a shared disk environment where multiple servers are using the same copy of the application.

Deploy Application from a Known Staging Area

Configure the `StagingDirName` attribute on each server to point to a well-known directory. You must place the actual EAR/JAR/WAR/RAR file in a directory having the name of the application within that directory.

Distribute Application Files to Managed Servers

If you set the `StagingMode` attribute to `stage`, WebLogic Server will copy the files from the source to the staging directory. To deploy using `stage` mode:

1. Configure the Managed Servers to use `stage` mode, and specify the staging directory each server uses.
2. Ensure that the files to deploy are available to the Administration Server for the domain—either copy the files to the Administration Server machine or place them on a shared filessytem available to the Administration Server machine.
3. Deploy the files to the Managed Servers using the Administration Console.

Deploy an Application Using external_stage Mode

`external_stage` mode requires that you either manually copy deployment files to Managed Servers or have an application or script copy the file for you. To deploy an application to Managed Servers using `external_stage` mode:

1. Configure the Managed Servers to use `stage` mode, and specify the staging directory each server uses.

2. In the staging area for the Managed Server, create a subdirectory with the deployment name you will use (for example, “mywar”), and copy the deployment files to that subdirectory.
3. Ensure that the files to deploy are available to the Administration Server for the domain—either copy the files to the Administration Server machine or place them on a shared filessytem available to the Administration Server machine.
4. Deploy the files to the Managed Servers using the Administration Console, using the same deployment name (“mywar”). The Managed Servers deploy using the deployment files you copied in Step 2 above.

Deployment Tools and Procedures

WebLogic Server deployment tools provide interfaces to the deployment API described in Deployment Management API.

The deployment instructions provided in this document presume that you have created a functional J2EE application that uses the correct directory structure and contains the appropriate deployment descriptors. Deployment descriptors, which are text files formatted with XML tags, describe the contents of the application directory or archive. The J2EE specifications define standard, portable deployment descriptors for J2EE applications and their components. BEA defines additional WebLogic-specific deployment descriptors for deploying an application and its components in the WebLogic Server environment. For more information, see XML Deployment Descriptors.

The following is a list of WebLogic Server deployment tools:

- weblogic.Deployer Utility
- wldeploy Ant Task
- WebLogic Server Administration Console
- WebLogic Builder
- Auto-Deployment (for Development Mode only)

weblogic.Deployer Utility

The `weblogic.Deployer` utility is new in WebLogic Server 7.0 and replaces the earlier `weblogic.deploy` utility, which has been deprecated. The `weblogic.Deployer` utility is a Java-based deployment tool that provides a command-line interface to the WebLogic Server deployment API. This utility was developed for administrators and developers who need to initiate deployment from the command line, a shell script, or any automated environment other than Java.

This section describes how to use the `weblogic.Deployer` utility to perform the following tasks:

- Deploying a New Application
- Deploying a New Application to a Cluster
- Redeploying an Entire Application
- Deploying a Module Newly Added to an EAR
- Redeploying Part of an Exploded Application, or Refreshing
- Deactivating an Application on All Active Targets, Making It Unavailable
- Reactivating a Deactivated Application
- Removing an Application from All Targeted Servers
- Cancelling a Deployment Task
- Listing All Deployment Tasks
- Deploying or Redeploying an Application to a Single Server
- Deploying an Application to an Additional Server

Deploying Using weblogic.Deployer Utility

To deploy an application or its components using the `weblogic.Deployer` utility:

1. Set up your local environment so that WebLogic Server classes are in your system `CLASSPATH` and the JDK is available. You can use the `setenv` script located in your server's `/bin` directory to set the `CLASSPATH`.

2. Use the following command syntax:

```
% java weblogic.Deployer [options]
[-activate|-deactivate|-remove|-cancel|-list] [files]
```

You can also list the specific `-files` in the archive that are to be deployed (or redeployed, or undeployed, or unprepared, or deactivated, or removed). The file list can include file names and directories relative to the root of the application. If you specify a directory, its entire subtree is deployed or redeployed.

weblogic.Deployer Actions and Options

Table 5-2 weblogic.Deployer Actions

Action	Description
activate	Deploys or redeploys the application specified by `-name` to the servers specified by `-targets`.
cancel	Attempts to cancel the task identified by `-id` if it is not yet completed.
deactivate	Deactivates the application on the target servers. Deactivation suspends the deployed components, leaving staged data in place in anticipation of subsequent reactivation. This command only works in the two-phase deployment protocol.
delete_files	Removes files specified in the file list and leaves the application activated. This is valid only for unarchived applications. You must specify target servers.
deploy	A convenient alias for `-activate`.
examples	Displays example usages of the tool.
help	Prints a help message.
list	Lists the status of the task identified by `-id`.

Action	Description
remove	Physically removes the application and any staged data from the target servers. The components are deactivated and the targets are removed from the applications configuration. If you remove the application entirely, the associated MBeans are also deleted from the system configuration. This command only works with the two-phase deployment model.
undeploy	A convenient alias for `-unprepare`.
unprepare	Deactivates and unloads classes for the application identified by `-name` on the target servers, leaving the staged application files in a state where they may be edited or quickly reloaded.
upload	Transfers the specified source file(s) to the administration server. Use this option when you are on a remote system and want to deploy an application that resides on the remote system. The application files are uploaded to the WebLogic Server administration server prior to distribution to named target servers.
version	Prints version information.

`weblogic.Deployer` options include:

Table 5-3 weblogic.Deployer Options

Option	Description
adminurl	`https://<server>:<port>` is the URL of the administration server. Default is `http://localhost:7001`.
debug	Turns on debug messages in the output log.
enforceClusterConstraints	Specifies whether a deployment to a cluster succeeds or fails when one or more members of the cluster are unavailable. Set this option to “true” to ensure that deployments succeed only when all members of the cluster are available.

Option	Description
external_stage	Indicates that user wants to copy the application to the servers' staging area externally on their own, or using a third-party tool. When specified, WLS looks for the application under `StagingDirectoryName (of target server)/applicationName`.
id	The task identifier `-id` is a unique identifier for the deployment task. You can specify an `-id` with the `-activate`, `-deactivate`, or `-remove` commands, and use it later as an argument to `-cancel` or `-list`. Make sure the `-id` is unique from all other existing deployment tasks. The system generates an `-id` if you do not specify one.
name	The application `-name` specifies the name of the application being deployed. This can be the name of an existing, configured application or the name to use when creating a new configuration.
nostage	Does not stage the application; instead, deploys it from its current location which you specify using the `-source` option. Defaults: `nostage` for admin server and `stage` for managed server targets.
nowait	Once the action is initiated, the tool prints the task id and exits. This is used to initiate multiple tasks and then monitor them later using the `-list` action.
output	Specify either `raw` or `formatted` to control the appearance of `weblogic.Deployer` output messages. Both output types contain the same information, but `raw` output does not contain embedded tabs. By default, `weblogic.Deployer` displays `raw` output.
password	Specifies the password on the command line. If you do not provide a password, you will be prompted for one.
remote	Signals that `weblogic.Deployer` is not running on the same machine as the administration server and that the source path should be passed through unchanged because it represents the path on the remote server.

Option	Description
source	Specifies the location of the archive, file or directory to be deployed. Use this option to set the application Path. The `source` option should reference the root directory or archive being deployed. If you are using it with the `upload` command, the source path is relative to the current directory. Otherwise, it is relative to the administration server root directory—the directory where the `config.xml` file resides.
stage	Indicates that application needs to be copied into the target servers staging area before deployment. Defaults: `nostage` for admin server, and `stage` for managed server targets. Sets the `stagingMethod` attribute on the application when it is created so that the application will always be staged. This value overrides the `stagingMethod` attribute on any targeted servers.
targets	Displays a comma-separated list of the targeted server and/or cluster names (`<server 1>,...<component>@<server N>`). Each target may be qualified with a J2EE component name. This enables different components of the archive to deployed on different servers. Default: For an application which is currently deployed, the default is all current targets. For a new application, it is deployed to the administration server, by default.
timeout	Seconds. Specifies the maximum time in seconds to wait for the completion of the deployment task. When the time expires, `weblogic.Deployer` prints out the current status of the deployment and exits.
user	User name.
userconfigfi le	Specifies the location of a user configuration file to use for the administrative username and password. Use this option, instead of the `-user` and `-password` options, in automated scripts or in situations where you do not want to have the password shown on-screen or in process-level utilities such as `ps`. Before specifying the `-userconfig` option, you must first generate the file using the `weblogic.Admin STOREUSERCONFIG` command.

Option	Description
userkeyfile	Specifies the location of a user key file to use for encrypting and decrypting the username and password information stored in a user configuration file (the `-userconfigfilefile` option). Before specifying the `-userkeyfile` option, you must first generate the key file using the `weblogic.Admin STOREUSERCONFIG` command.
verbose	Displays additional progress messages.

Example Uses of the weblogic.Deployer Utility

Below are example usages of the `weblogic.Deployer` utility.

Deploying a New Application

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-source /myapp/app.ear -targets server1,server2 -activate
```

Deploying a New Application to a Cluster

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-source /myapp/app.ear -targets cluster1 -activate
-enforceClusterConstraints
```

Redeploying an Entire Application

```
java weblogic.Deployer -source /myapp/app.ear -adminurl
http://admin:7001 -name app -activate
```

Notes: Ensure that you specify the –source option or provide the list of updated files. Without this, the operation will have no effect as the system will assume that nothing has been changed in the application.

When redeploying a Web Application, the system defaults to deploying the application on the Administration Server. To change the settings, use the -source and -targets options.

Deploying a Module Newly Added to an EAR

If you have added the module `newmodule.war` to the deployed application `myapp.ear` and updated the module in the `application.xml` file, you can deploy `newmodule.war` in `myapp.ear` using the following:

```
java weblogic.Deployer -username myname -password mypassword
-name myapp.ear -activate -targets newmodule.war@myserver
-source /myapp/myapp.ear
```

Note that this command will deploy the new module without redeploying the other modules in the application.

Redeploying Part of an Exploded Application, or Refreshing

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-activate jsps/login.jsp
```

where `jsps` is a directory in the top level of the exploded web application. Note that partial redeployment is only supported on exploded `WAR` files. The path is relative to the root of the application as originally deployed. For example, if you modified the `login.jsp` file in a working directory, you would need to first copy the updated file into the appropriate source directory of the exploded application before entering the `weblogic.Deployer` command.

Deactivating an Application on All Active Targets, Making It Unavailable

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-deactivate
```

Reactivating a Deactivated Application

```
 java weblogic.Deployer -adminurl http://7001 -name app
-activate
```

Removing an Application from All Targeted Servers

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-targets server -remove
```

Cancelling a Deployment Task

```
java weblogic.Deployer -adminurl http://admin:7001 -cancel -id
tag
```

Listing All Deployment Tasks

```
java weblogic.Deployer -adminurl http://admin:7001 -list
```

Deploying or Redeploying an Application to a Single Server

```
java weblogic.Deployer -activate -name ArchivedEarJar -source
C:/MyApps/JarEar.ear -target server1
```

Deploying an Application to an Additional Server

```
java weblogic.Deployer -activate -name ArchivedEarJar -target
server2
```

wldeploy Ant Task

The `wldeploy` Ant task enables you to perform `weblogic.Deployer` functions using attributes specified in an Ant .xml file. You can use `wldeploy` along with other WebLogic Server Ant tasks to create a single Ant build script that:

- Creates, starts, and configures a new WebLogic Server domain, using the `wlserver` and `wlconfig` Ant tasks.
- Deploys a compiled application to the newly-created domain, using the `wldeploy` Ant task.

See Using Ant Tasks to Configure a WebLogic Server Domain in the *Administration Guide* for more information about `wlserver` and `wlconfig`.

Note: The WebLogic Server Ant tasks are incompatible with Ant versions prior to 1.5. Also, if you use a version of Ant that is not included with WebLogic Server, you must specify the `wldeploy` task definition in your `build.xml` file, as described in "Basic Steps for Using wldeploy" on page 5-18.

Basic Steps for Using wldeploy

To use the `wldeploy` Ant task:

1. Set your environment.

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

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

2. In the staging directory, create the Ant build file (`build.xml` by default). If you want to use an Ant installation that is different from the one installed with WebLogic Server, start by defining the `wldeploy` Ant task definition:

```
<taskdef name="wldeploy"
classname="weblogic.ant.taskdefs.management.WLDeploy"/>
```

3. If necessary, add task definitions and calls to the `wlserver` and `wlconfig` tasks in the build script to create and start a new WebLogic Server domain. See Using Ant Tasks to Configure a WebLogic Server Domain in the *WebLogic Server Command Reference* for information about `wlserver` and `wlconfig`.

4. Add a call to `wldeploy` to deploy your application to one or more WebLogic Server instances or clusters. See "Sample build.xml Files for wldeploy" on page 5-19 and "wldeploy Ant Task Reference" on page 5-20.

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

```
prompt> ant
```

Sample build.xml Files for wldeploy

The following output shows a `wldeploy` target that deploys an application to a single WebLogic Server instance:

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

wldeploy Ant Task Reference

The following table describes the attributes of the `wldeploy` Ant task. For more information about the definition of various terms, see "weblogic.Deployer Actions and Options" on page 5-12.

Table 5-4 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type	Required?
action	The deployment action to perform. Valid values are `activate`, `deactivate`, `remove`, `cancel`, `list` and `unprepare`.	String	No
adminurl	The URL of the Administration Server.	String	No
debug	Enable `wldeploy` debugging messages.	boolean	No
id	Identification used for obtaining status or cancelling the deployment.	String	No
name	The deployment name for the deployed application.	String	No
nostage	Specifies whether the deployment uses nostage deployment mode.	boolean	No
nowait	Specifies whether wldeploy returns immediately after making a deployment call (by deploying as a background task).	boolean	No
user	The administrative username.	String	No

Table 5-4 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type	Required?
password	The administrative password. To avoid having the plain text password appear in the build file or in process utilities such as `ps`, first store a valid username and encrypted password in a configuration file using the `weblogic.Admin STOREUSERCONFIG` command. Then omit both the `username` and `password` attributes in your Ant build file. When the attributes are omitted, `wldeploy` attempts to login using values obtained from the default configuration file. If you want to obtain a username and password from a non-default configuration file and key file, use the `userconfigfile` and `userkeyfile` attributes with `wldeploy`.	String	No
remote	Specifies whether the server is located on a different machine. This affects how filenames are transmitted.	boolean	No
source	The source file to deploy.	File	No
targets	The list of target servers to deploy to.	String	No
timeout	The maximum time to wait for a deployment to succeed.	int	No
userconfigfile	Specifies the location of a user configuration file to use for obtaining the administrative username and password. Use this option, instead of the `user` and `password` attributes, in your build file when you do not want to have the plain text password shown in-line or in process-level utilities such as `ps`. Before specifying the `userconfigfile` attribute, you must first generate the file using the `weblogic.Admin STOREUSERCONFIG` command as described in STOREUSERCONFIG in the *WebLogic Server Command-Line Reference*.	File	No

Table 5-4 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type	Required?
userkeyfile	Specifies the location of a user key file to use for encrypting and decrypting the username and password information stored in a user configuration file (the `userconfigfile` attribute). Before specifying the `userkeyfile` attribute, you must first generate the key file using the `weblogic.Admin STOREUSERCONFIG` command.	File	No
verbose	Specifies whether `wldeploy` displays verbose output messages.	boolean	No
failonerror	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. This attribute is set to true by default.	Boolean	No

WebLogic Server Administration Console

This section discusses deployment tasks performed through the Administration Console. The Console supports the same functionality as the `weblogic.Deployer` utility. It allows deployers to submit new or updated applications and to query the status and remove pending deployments.

Configuring J2EE Applications for Deployment Using the Administration Console

To configure a J2EE Application using the WebLogic Server Administration Console:

1. Start the WebLogic Server Administration Console.
2. Select the Domain in which you will be working.
3. In the left pane of the Console, click Deployments.
4. In the left pane of the Console, click Applications. A table is displayed in the right pane of the Console showing all the deployed J2EE Applications.

5. Select the Configure a new Application option.
6. Locate the archive file (WAR, EAR, RAR, JAR) to configure.

   Note: You can also configure an exploded application or component directory. Note that WebLogic Server deploys all components it finds in and below the specified directory.

7. Click [select] to the left of a directory or file to choose it and proceed to the next step.
8. Select a Target Server from among Available Servers.
9. Enter a name for the Application in the provided field.
10. Click Configure and Deploy. The Console will display the Deploy panel, which lists deployment status and deployment activities for the J2EE Application.
11. Using the available tabs, enter the following information:
    - Configuration—Edit the staging mode and enter the deployment order.
    - Targets—Indicate the Targets-Server for this configured J2EE Application by moving the server from the Available list to the Chosen list.

    Note: If you do not select any targets, the application will just be configured and not deployed. You can modify and deploy this later by accessing it in the `mydomain/applications` directory.

    - Deploy/Undeploy—Deploy the J2EE Application to all or selected targets or undeploy it from all or selected targets; undeploy the J2EE Application. Deploy/Undeploy are toggle options.
    - Monitoring—Enable session monitoring for the J2EE Application.
    - Notes—Enter notes related to the J2EE Application.

Deploying J2EE Applications with the Administration Console

To deploy a J2EE application using the WebLogic Server Administration Console:

1. Expand the Deployments node in the left pane.
2. Right-click on the Applications node.
3. Select Configure a New Application.

4. Locate the archive (WAR, EAR, RAR, JAR) or the directory containing the exploded application to configure.
5. Click [select] to the left of a directory or file to choose it and proceed to the next step. If you specify a directory, WebLogic Server will deploy all components it finds in and below the specified directory.
6. Select a Target Server from among Available Servers.
7. Enter a name for the J2EE Application in the provided field.
8. Click Configure and Deploy. The Console will display the Deploy panel, which lists deployment status and deployment activities for the J2EE Application.
9. Use the Deploy button to deploy the J2EE Application to all or selected targets or undeploy it from all or selected targets.
10. Test your J2EE Application by accessing a resource through a Web browser. Access resources with a URL constructed as follows:

    http://myServer:myPort/myApp/resource

    Where:

    - myServer is the name of the machine hosting WebLogic Server.
    - myPort is the port number where WebLogic Server is listening for requests.
    - myApp is the name of the J2EE Application archive file (myApp.ear, for instance) or the name of a directory containing the J2EE Application.
    - resource is the name of a resource such as a JSP, HTTP servlet, or HTML page.

Viewing Deployed Components with the Administration Console

To view a deployed component in the Administration Console:

1. In the Administration Console under Deployments, select the <component> in the left panel.
2. View a list of deployed components in the Deployments table in the right pane.

Undeploying Components with the Administration Console

To undeploy a deployed component from the WebLogic Server Administration Console:

1. In the Administration Console under Deployments, select the component in the left panel.
2. In the component Deployments table, select the component to undeploy.
3. Click Apply.

Undeploying a component does not remove the <component> name from WebLogic Server. The component remains undeployed for the duration of the Server session, as long as you do not change it once it has been undeployed. You cannot re-use the deployment name with the deploy argument until you reboot the server. You can re-use the deployment name to update the deployment, as described in the following section.

Updating Applications with the Administration Console

To update a deployed J2EE application:

1. In the Console, click Deployments.
2. Click the Applications option.
3. In the displayed table, click the name of the application you wish to update.
4. Update the Application Name and Deployed status as needed.

   Note: An application becomes unavailable to clients during redeployment. For this reason, redeployment is not recommended for use in a production environment.

5. Click Apply.

To add a module to a deployed application and deploy the added module:

1. Add the module by redeploying the application, using steps 1-5 above.
2. Click the module name in the table in the Deployments >Applications tab.
3. Select the Targets tab.

4. Move the desired server from the Available area to the Chosen area, and click Apply.

WebLogic Builder

WebLogic Builder is a WebLogic Server tool for generating and editing deployment descriptors for J2EE applications. It can also deploy applications to single servers.

See WebLogic Builder.

Auto-Deployment

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

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

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

Enabling and Disabling Auto-Deployment

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

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

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

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

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

`STARTMODE` = false enables deployment mode

`STARTMODE` = true enables production mode

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

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

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

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

Auto-Deploying Applications

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

Undeploying and Redeploying Archived Applications

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

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

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

Redeploying Applications in Exploded Format

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

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

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

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

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

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

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

   Note: The `REDEPLOY` file works only for an entire deployed application or a deployed standalone module. If you have deployed an exploded Enterprise Application, the `REDEPLOY` file controls redeployment for the entire application—not for individual modules (for example, a Web Application) within the Enterprise Application. If you deploy a Web Application by itself as an exploded archive directory, the `REDEPLOY` file controls redeployment for the entire Web Application.

2. To update the exploded application, copy the updated files over the existing files in that directory.

3. After copying the new files, modify the `REDEPLOY` file in the exploded directory to alter its timestamp.

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

Deployment Management API

A deployment task is initiated through a `DeployerRuntimeMBean`—a singleton (an object for which only one instance exists) that resides on a WebLogic Administration Server. `DeployerRuntimeMBean` provides methods for activating, deactivating, and removing an application. These methods return a `DeploymentTaskRuntimeMBean` that encapsulates the request and provides the means for tracking its progress. `DeploymentTaskRuntimeMBean` provides ongoing status of the request through TargetStatus objects, one per target.

The WebLogic Server deployment management API is defined by the following WebLogic Server MBeans:

- `DeployerRuntimeMBean`—programmatic interface to deployment requests. Deployment requests provided through the `DeployerRuntimeMBean` manifest the configuration state into the application and appropriate component configuration MBeans. These MBeans persist the deployment state of applications in the WebLogic Server domain.
- `DeploymentTaskRuntimeMBean`—interface for encompassing deployment tasks.

The deployment management API is asynchronous. The client must poll the status or utilize `ApplicationMBean` notifications to determine when the task is complete.

For more information about WebLogic Server deployment management APIs, see the `weblogic.management.deploy` Javadoc.

Best Practices for Application Deployment

The following are some best practices for deploying applications.

Single Server Development

In an iterative development environment, it is most efficient to maintain all files in an exploded form and deploy the application directly from its source location.

If you are working in a single server environment (deploying only to the administration server), you can use directory-based auto deployment. In other words, you can simply place the exploded application in the `mydomain/applications` directory to deploy it. For more information, see Auto-Deployment in this document.

Testing Changes to Web Applications or Web Services

- You can change JSPs or any static data files in the application under the `mydomain/applications` directory, and view your changes in the application.
- You can also directly compile updated servlet classes into a web application under the `mydomain/applications` directory (for instance, `mydomain/applications/exploded_web_app_dir/WEB-INF/classes`). The new classes are incorporated in the Web application; no additional steps are required.
- To incorporate changes made to the Web application deployment descriptors, modify the `WEB-INF/REDEPLOY` file in the `mydomain/applications` directory, so that the auto deployer detects the change. This redeploys the Web application.

Testing Changes to EJBs and Resource Adapters

To incorporate changes made to the EJB or Resource Adapter deployment descriptors, modify the `META-INF/REDEPLOY` file in the `mydomain/applications` directory, so that the auto deployer detects the change. This redeploys the EJB or Resource Adapter.

Note: If the EJB is part of an enterprise application, the entire application is redeployed.

Multiple Server Development

If you are working in a multiple server environment, it is recommended that you use the `weblogic.Deployer` tool to deploy applications.

Testing Changes in a Multiple Server Environment

If you have made any changes to application files, you must communicate these changes to the server using the `weblogic.Deployer` tool. This allows the changes to be incorporated into the deployed application.

The following steps illustrate how you communicate changes made to a Web application to the server. The steps are identical for any type of application. However, you should note that if an EJB is part of an enterprise application, the entire application and application components are redeployed.

1. Deploy the Web application using the Administration Console or as follows:

   ```
   java weblogic.Deployer -adminurl http://adminAddr:7001 -name
   webapp -activate -source /myapp/webapp -targets
   managedserver1,managedserver2
   ```

2. Make needed changes to the JSP: /myapp/webapp/jsps/login.jsp

3. Apply the changes as follows:

   ```
   java weblogic.Deployer -adminurl http://adminAddr:7001 -name
   webapp -activate jsps/login.jsp
   ```

In the above example, `login.jsp` is distributed and incorporated to all servers where the Web application is deployed.

File Structures for Exploded Applications

For more information about packaging deployable units, see WebLogic Server Application Packaging and Classloading.

If a directory contains multiple modules but no overall application descriptor file, each module should be contained in its own directory and have its own descriptor files. In other words, in non-archived applications each module must be independent of the other modules, just as in an archive.

```
sourceDirectory\

               \Module1
                     WEB-INF\
                             web.xml
                             Module1FilesDir
```

```
               \Module2
                     META-INF\
                             ejb-jar.xml
                             Module2FilesDir
```

For exploded non-EAR deployments, the source directory should always be either A or B:

```
A.

 \Module1
         WEB-INF\
                web.xml
                Module1FilesDir
```

B.

```
 \Module1
         WEB-INF\
                web.xml
                Module1FilesDir

 \Module2
         META-INF\
                ejb-jar.xml
                Module2FilesDir
 \Module3
         META-INF\
                ra.xml
                Module3FilesDir
```

The following directory structures are not supported and do not make sense:

```
sourceDirectory/

       META-INF/ejb-jar.xml

       WEB-INF/web.xml

       webfiles/

       ejbfiles/


moduledir/

       WEB-INF/web.xml
```

```
        webfiles

        ejb.jar

moduledir/

        WEB-INF/web.xml

        web1files

        web2/WEB-INF/web.xml

        web2/webfiles
```

Staging Mode

Use the system defaults of `-nostage` for administration server and `-stage` for managed servers, unless:

- You want to use third-party solutions to manage file copying and distribution from the administration server to the managed server machines.
- You want to use a shared file system for sharing the source of an application between different servers in a domain.

Auto-Deployment

You should only use auto-deployment in development setups for single server deployments.

Exploded Enterprise Applications

Exploded deployments with multiple modules should always have an application descriptor defined in `META-INF/application.xml`.

Partial Redeployment

If you redeploy a module or file to which other modules in the application have references, you must also redeploy the referencing modules.

Sharing Classes between Components That Are Part of an Enterprise Application

To share classes between components that are part of an Enterprise Application, use the `MANIFEST` classpath. A JAR utility containing the shared classes is packed in the EAR next to the other component archives. Each component needing to use these classes creates a Class-Path entry in a file named `META-INF/MANIFEST.MF` within the component's archive. This scheme is part of the J2EE standard and should be used if you require portability between application servers.

For more information, refer to "Manifest Class-Path" on page 3-10.

Using WebLogic Server 6.x Deployment Protocol

By default, the two-phase deployment protocol is used for deploying new applications by all available deployment tools. The current administration server still supports the WebLogic Server 6.x deployment protocol, and this protocol is used when:

- Configured applications do not specify the two-phase deployment protocol by setting `ApplicationMBean.TwoPhase=false`.
- The application contains multiple modules and is not an EAR.

See Deploying Applications in the *WebLogic Server 6.1 Administration Guide*.

Note: When using the WebLogic Server 6.x deployment protocol, a server instance creates a `.wlnotdelete` directory to manage the deployment process. You should not delete this directory or its contents.

Updating to Two Phase Deployment

To configure an application that uses 6.x protocol to start using the two-phase protocol, remove the application from the domain—removing its configuration—and then re-activate the application, as follows:

1. Remove the application using `weblogic.Deployer`. Enter a command in the following form:

```
java weblogic.Deployer -adminurl http://admin:7001 -name app
-targets server -remove
```

2. Reactivate the application using `weblogic.Deployer`. Enter a command in the following form:

```
java weblogic.Deployer -activate -name ArchivedEarJar -source
C:/MyApps/JarEar.ear -target server1
```

The application will redeploy using the new protocol.

Additional Deployment Documentation

For more information on WebLogic Server deployment, see the following documentation:

Document	Deployment Topics
WebLogic Builder	How to use WebLogic Builder to edit and generate XML deployment descriptor files for J2EE applications and their components.
Administration Console Online Help	How to use the Administration Console for deployment tasks.

Document	Deployment Topics
Understanding Cluster Configuration and Application Deployment	How to deploy to clustered servers.
Programming WebLogic EJBs	How to deploy WebLogic Server EJBs.
Programming WebLogic J2EE Connectors	How to deploy WebLogic Server J2EE Connectors.
Assembling and Configuring Web Applications	How to deploy Weblogic Server Web Applications.
Programming WebLogic JSP	How to deploy applets from JSP.
WebLogic Server Application Packaging and Classloading	How to package WebLogic Server application components.

CHAPTER

6 Programming Topics

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

Logging Messages

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

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

Using Threads in WebLogic Server

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

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

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

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

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

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

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

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

Using JavaMail with WebLogic Server Applications

WebLogic Server includes the JavaMail API version 1.1.3 reference implementation from Sun Microsystems. Using the JavaMail API, you can add email capabilities to your WebLogic Server applications. JavaMail provides access from Java applications to Internet Message Access Protocol (IMAP)- and Simple Mail Transfer Protocol (SMTP)-capable mail servers on your network or the Internet. It does not provide mail server functionality; so you must have access to a mail server to use JavaMail.

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

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

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

About JavaMail Configuration Files

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

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

Configuring JavaMail for WebLogic Server

To configure JavaMail for use in WebLogic Server, you create a Mail Session in the WebLogic Server Administration Console. This allows server-side components and applications to access JavaMail services with JNDI, using Session properties you preconfigure for them. For example, by creating a Mail Session, you can designate the mail hosts, transport and store protocols, and the default mail user in the Administration Console so that components that use JavaMail do not have to set these properties. Applications that are heavy email users benefit because WebLogic Server creates a single Session object and makes it available via JNDI to any component that needs it.

1. In the Administration Console, click on the Mail node in the left pane of the Administration Console.

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

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

`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.

Property	Description	Default

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

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

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

Sending Messages with JavaMail

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

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

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

2. Look up the Mail Session in JNDI:

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

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

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

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

   ```
   Message msg = new MimeMessage(session2);
   msg.setFrom();
   msg.setRecipients(Message.RecipientType.TO,
                     InternetAddress.parse(to, false));
   msg.setSubject(subject);
   msg.setSentDate(new Date());
   // Content is stored in a MIME multi-part message
   // with one body part
   MimeBodyPart mbp = new MimeBodyPart();
   mbp.setText(messageTxt);

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

5. Send the message.

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

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

Reading Messages with JavaMail

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

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

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

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

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

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

2. Look up the Mail Session in JNDI:

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

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

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

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

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

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

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

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

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

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

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

Programming Applications for WebLogic Server Clusters

JSPs and Servlets that will be deployed to a WebLogic Server cluster must observe certain requirements for preserving session data. See "Using WebLogic Server Clusters" for more information.

EJBs deployed in a WebLogic Server cluster have certain restrictions based on EJB type. See "The WebLogic Server EJB Container" in "Programming WebLogic Enterprise JavaBeans" for information about the capabilities of different EJB types in a cluster. EJBs can be deployed to a cluster by setting clustering properties in the EJB deployment descriptor. "weblogic-ejb-jar.xml Deployment Descriptors" in "Programming WebLogic Enterprise JavaBeans" describes the XML deployment elements relevant for clustering.

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

A Application Deployment Descriptor Elements

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

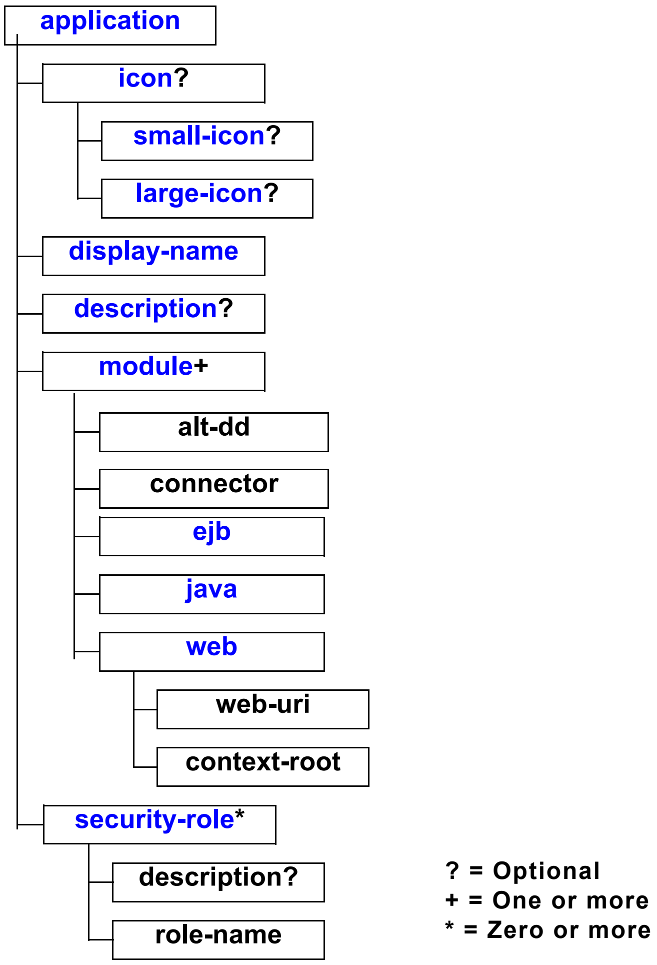
application.xml Deployment Descriptor Elements

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

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

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

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



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

application

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

icon

Optional. The `icon` element specifies the locations of small and large images that represent the application in a GUI tool. This element is not currently used by WebLogic Server.

small-icon

Optional. Specifies the location for a small (16x16 pixel) `.gif` or `.jpg` image used to represent the application in a GUI tool. Currently, this is not used by WebLogic Server.

large-icon

Optional. Specifies the location for a large (32x32 pixel) `.gif` or `.jpg` image used to represent the application in a GUI tool. Currently, this element is not used by WebLogic Server.

display-name

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

description

The optional description element provides descriptive text about the application.

module

The `application.xml` deployment descriptor contains one `module` element for each module in the Enterprise Archive file. Each `module` element contains an `ejb`, `java`, or `web` element that indicates the module type and location of the module within the application. An optional `alt-dd` element specifies an optional URI to the post-assembly version of the deployment descriptor.

alt-dd

Specifies an optional URI to the post-assembly version of the deployment descriptor file for a particular J2EE module. The URI must specify the full pathname of the deployment descriptor file relative to the application's root directory. If you do not specify `alt-dd`, the deployer must read the deployment descriptor from the default location and file name required by the respective component specification.

connector

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

ejb

Defines an EJB module in the application file. Contains the path to an EJB JAR file in the application.

Example:

```
<ejb>petStore_EJB.jar</ejb>
```

java

Defines a client application module in the application file.

Example:

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

web

Defines a Web application module in the `application.xml` file. The `web` element contains a `web-uri` element and a `context-root` element. If you do not declare a value for the `context-root`, then the basename of the `web-uri` element is used as the context path of the Web application. (Note that the context path must be unique in a given Web server. More than one Web application may be using the same Web server, so you must avoid having context path clashes across multiple applications.)

web-uri

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

context-root

Specifies a context root for the Web application.

Example:

```
<web>
  <web-uri>petStore.war</web-uri>
  <context-root>estore</context-root>
</web>
```

security-role

The `security-role` element contains the definition of a security role which is global to the application. Each `security-role` element contains an optional `description` element, and a `role-name` element.

description

Optional. Text description of the security role.

role-name

Defines the name of a security role or principal that is used for authorization within the application. Roles are mapped to ProductName users or groups in the `weblogic-application.xml` deployment descriptor.

Example:

```
<security-role>
  <description>the gold customer role</description>
  <role-name>gold_customer</role-name>
</security-role>
<security-role>
  <description>the customer role</description>
  <role-name>customer</role-name>
</security-role>
```

weblogic-application.xml Deployment Descriptor Elements

The following sections describe the `weblogic-application.xml` file. The `weblogic-application.xml` file is the BEA WebLogic Server-specific deployment descriptor extension for the `application.xml` deployment descriptor from Sun Microsystems. This is where you configure features such as application-scoped JDBC Pools and EJB Caching.

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

```
<!DOCTYPE weblogic-application PUBLIC "-//BEA Systems, Inc.//DTD
WebLogic Application 7.0.0//EN"

"http://www.bea.com/servers/wls700/dtd/weblogic-application_1_0.d
td;">
```

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

weblogic-application

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

ejb

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

entity-cache

One or more. The `entity-cache` element is used to define a named application level cache that is used to cache entity EJB instances at runtime. Individual entity beans refer to the application-level cache that they must use, referring to the cache name. There is no restriction on the number of different entity beans that may reference an individual cache.

Application-level caching is used by default whenever an entity bean does not specify its own cache in the `weblogic-ejb-jar.xml` descriptor. Two default caches named `ExclusiveCache` and `MultiVersionCache` are used for this purpose. An application may explicitly define these default caches to specify non-default values for their settings. Note that the caching-strategy cannot be changed for the default caches. By default, a cache uses `max-beans-in-cache` with a value of 1000 to specify its maximum size.

Example:

```
<entity-cache>

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

       <max-cache-size>

              <megabytes>50</megabytes>

       </max-cache-size>

</entity-cache>
```

entity-cache-name

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

Example:

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

max-beans-in-cache

Optional. The `max-beans-in-cache` element specifies the maximum number of entity beans that are allowed in the cache. If the limit is reached, beans may be passivated. This mechanism does not take into account the actual amount of memory that different entity beans require. This element can be set to a value of 1 or greater.

Default Value: `1000`

max-cache-size

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

- `bytes | megabytes`—The size of an entity cache in terms of memory size, expressed in bytes or megabytes. Used in the `max-cache-size` element.

caching-strategy

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

The `caching-strategy` element can only have one of the following values:

- `Exclusive`—Caches a single bean instance in memory for each primary key value. This unique instance is typically locked using the EJB container's exclusive locking when it is in use, so that only one transaction can use the instance at a time.
- `MultiVersion`—Caches multiple bean instances in memory for a given primary key value. Each instance can be used by a different transaction concurrently.

Default Value: `MultiVersion`

Example:

```
<caching-strategy>Exclusive</caching-strategy>
```

start-mdbs-with-application

```
Optional. Allows you to configure the EJB container to start Message
Driven BeanS (MDBS) with the application. If set to true, the
container starts MDBS as part of the application. If set to false,
the container keeps MDBS in a queue and the server starts them as
soon as it has started listening on the ports.
```

xml

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

parser-factory

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

saxparser-factory

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

Default Value: Server XML Registry setting

document-builder-factory

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

Default Value: Server XML Registry setting

transformer-factory

Optional. The `transformer-factory` element allows you to set the Transformer Engine for the style sheet processing required in this application only. If you do not specify a value for this element, the value configured in the Server XML Registry is used.

Default value: Server XML Registry setting.

entity-mapping

Zero or more. The `entity-mapping` element is used to specify entity mapping. This mapping determines the alternative entity URI for a given public or system ID. The default place to look for this entity URI is the `lib/xml/registry` directory.

entity-mapping-name

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

public-id

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

system-id

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

entity-uri

Optional. The `entity-uri` element specifies the entityuri for the mapped entity.

when-to-cache

Optional. Legal values are:

- cache-on-reference
- cache-at-initialization
- cache-never

The default value is `cache-on-reference`.

cache-timeout-interval

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

jdbc-connection-pool

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

data-source-name

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

connection-factory

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

factory-name

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

connection-properties

Optional. The `connection-properties` element specifies the connection parameters that define overrides for default connection factory settings.

- `user-name`—Optional. The `user-name` element is used to override `UserName` in the `JDBCDataSourceFactoryMBean`.
- `url`—Optional. The `url` element is used to override `URL` in the `JDBCDataSourceFactoryMBean`.
- `driver-class-name`—Optional. The `driver-class-name` element is used to override `DriverName` in the `JDBCDataSourceFactoryMBean`.
- `connection-params`—Zero or more.
  - `parameter+` (`param-value`, `param-name`)—One or more

pool-params

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

size-params

Optional. The `size-params` element defines parameters that affect the number of connections in the pool.

- `initial-capacity`—Optional. The `initial-capacity` element defines the number of physical database connections to create when the pool is initialized. The default value is `1`.
- `max-capacity`—Optional. The `max-capacity` element defines the maximum number of physical database connections that this pool can contain. Note that the JDBC Driver may impose further limits on this value. The default value is `1`.
- `capacity-increment`—Optional. The `capacity-increment` element defines the increment by which the pool capacity is expanded. When there are no more available physical connections to service requests, the pool creates this number of additional physical database connections and adds them to the pool. The pool ensures that it does not exceed the maximum number of physical connections as set by `max-capacity`. The default value is `1`.

- `shrinking-enabled`—Optional. The `shrinking-enabled` element indicates whether or not the pool can shrink back to its `initial-capacity` when connections are detected to not be in use.
- `shrink-period-minutes`—Optional. The `shrink-period-minutes` element defines the number of minutes to wait before shrinking a connection pool that has incrementally increased to meet demand. The `shrinking-enabled` element must be set to `true` for shrinking to take place.

xa-params

Optional. The `xa-params` element defines the parameters for the XA DataSources.

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

statement cache. The size of the cache is a number of prepared statements created from a particular connection and stored in the cache for further use. Setting the size of the prepared statement cache to `0` turns it off.

- `keep-logical-conn-open-on-release`—Optional. Boolean. Set the `keep-logical-conn-open-on-release` element to `true`, to keep the logical JDBC connection open when the physical XA connection is returned to the XA connection pool. The default value is `false`.
- `local-transaction-supported`—Optional. Boolean. Set the `local-transaction-supported` to `true` if the XA driver supports SQL with no global transaction; otherwise, set it to `false`. The default value is `false`.
- `resource-health-monitoring-enabled`—Optional. Set the `resource-health-monitoring-enabled` element to `true` to enable JTA resource health monitoring for this connection pool.

login-delay-seconds

Optional. Integer value. The `login-delay-seconds` element sets the number of seconds to delay before creating each physical database connection. Some database servers cannot handle multiple requests for connections in rapid succession. This property allows you to build in a small delay to let the database server catch up. This delay occurs both during initial pool creation and during the lifetime of the pool whenever a physical database connection is created.

leak-profiling-enabled

Optional. The `leak-profiling-enabled` element enables JDBC connection leak profiling. A connection leak occurs when a connection from the pool is not closed explicitly by calling the `close()` method on that connection. When connection leak profiling is active, the pool stores the stack trace at the time the connection object is allocated from the pool and given to the client. When a connection leak is detected (when the connection object is garbage collected), this stack trace is reported.

This element uses extra resources and will likely slowdown connection pool operations, so it is not recommended for production use.

connection-check-params

Optional. The `connection-check-params` element defines whether, when, and how connections in a pool is checked to make sure they are still alive.

- `table-name`—Optional. The `table-name` element defines a table in the schema that can be queried.
- `check-on-reserve-enabled`—Optional. If the `check-on-reserve-enabled` element is set to `true`, then the connection will be tested each time before it is handed out to a user.
- `check-on-release-enabled`—Optional. If the `check-on-release-enabled` element is set to `true`, then the connection will be tested each time a user returns a connection to the pool.
- `refresh-minutes`—Optional. If the `refresh-minutes` element is defined, a trigger is fired periodically (based on the number of minutes specified). This trigger checks each connection in the pool to make sure it is still valid.

driver-params

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

statement

Optional.

- `profiling-enabled`—Optional. `profiling-enabled` boolean. The `profiling-enabled` element enables the running of JDBC SQL roundtrip profiling. When enabled, SQL statement text, execution time, and other metrics are stored externally for further analysis. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. The default value is `false`.

prepared-statement

Optional. `profiling-enabled` boolean. The `prepared-statement` element enables the running of JDBC prepared statement cache profiling. When enabled, prepared statement cache profiles are stored in external storage for further analysis. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. The default value is `false`.

- `profiling-enabled`—Optional.
- `cache-profiling-threshold`—Optional. The `cache-profiling-threshold` element defines a number of statement

requests after which the state of the prepared statement cache is logged. This element minimizes the ouput volume. This is a resource-consuming feature, so it is recommended that you turn it off on a production server.

- `cache-size`—Optional. The `cache-size` element returns the size of the prepared statement cache. The size of the cache is a number of prepared statements created from a particular connection and stored in the cache for further use.
- `parameter-logging-enabled`—Optional. During SQL roundtrip profiling it is possible to store values of prepared statement parameters. The `parameter-logging-enabled` element enables the storing of statement parameters. This is a resource-consuming feature, so it is recommended that you turn it off on a production server.
- `max-parameter-length`—Optional. During SQL roundtrip profiling it is possible to store values of prepared statement parameters. The `max-parameter-length` element defines maximum length of the string passed as a parameter for JDBC SQL roundtrip profiling. This is a resource-consuming feature, so you should limit the length of data for a parameter to reduce the output volume.

row-prefetch-enabled

Optional

row-prefetch-size

Optional

stream-chunk-size

Optional

acl-name

Optional

application-param

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

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

CHAPTER

B Client Application Deployment Descriptor Elements

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

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

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

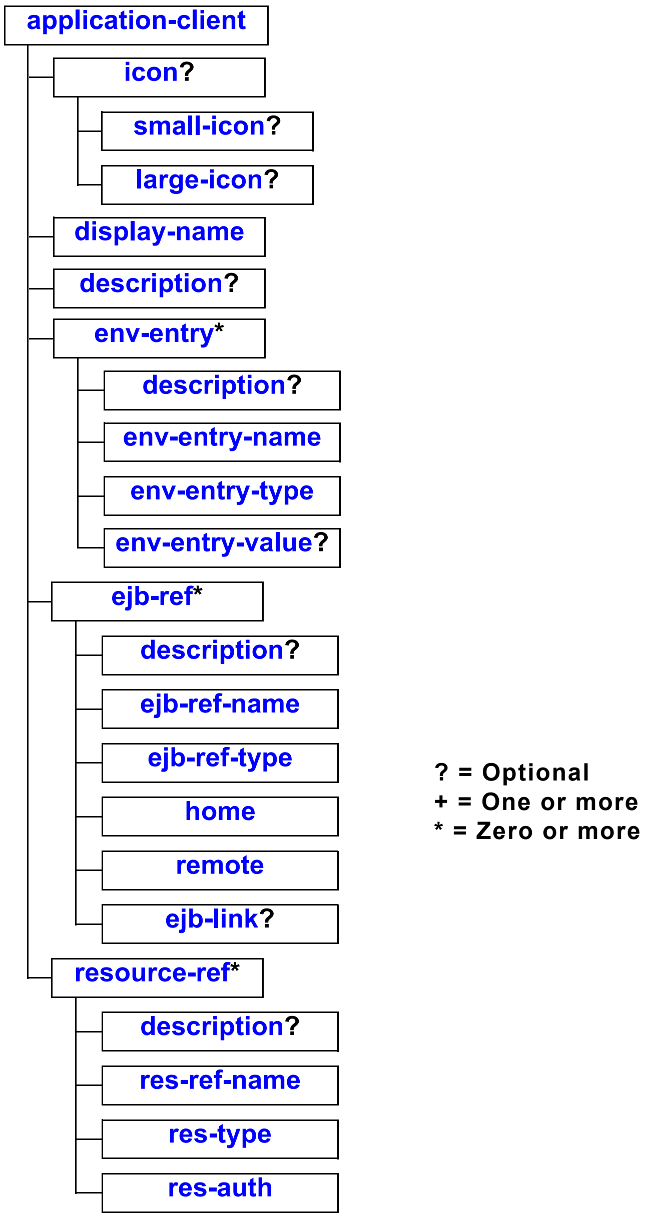
- "application-client.xml Deployment Descriptor Elements" on page B-2
- "WebLogic Run-time Client Application Deployment Descriptor" on page B-7

application-client.xml Deployment Descriptor Elements

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

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

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



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

application-client

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

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

icon

Optional. The `icon` element specifies the locations of small and large images that represent the application in a GUI tool. This element is not currently used by WebLogic Server.

small-icon

Optional. Specifies the location for a small (16x16 pixel) `.gif` or `.jpg` image used to represent the application in a GUI tool. Currently, this is not used by WebLogic Server.

large-icon

Optional. Specifies the location for a large (32x32 pixel) `.gif` or `.jpg` image used to represent the application in a GUI tool. Currently, this element is not used by WebLogic Server.

display-name

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

description

Optional. The `description` element provides a description of the client application.

env-entry

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

description

Optional. The `description` element contains a description of the particular environment entry.

env-entry-name

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

env-entry-type

The `env-entry-type` element contains the fully-qualified Java type of the environment entry. The possible values are: `java.lang.Boolean`, `java.lang.String`, `java.lang.Integer`, `java.lang.Double`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Long`, and `java.lang.Float`.

env-entry-value

Optional. The `env-entry-value` element contains the value of a client application's environment entry. The value must be a String that is valid for the constructor of the specified `env-entry-type`.

ejb-ref

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

description

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

ejb-ref-name

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

ejb-ref-type

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

home

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

remote

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

ejb-link

The `ejb-link` element specifies that an EJB reference is linked to an enterprise JavaBean in the J2EE application package. The value of the `ejb-link` element must be the name of the `ejb-name` of an EJB in the same J2EE application.

resource-ref

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

description

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

res-ref-name

The `res-ref-name` element specifies the name of the resource factory reference name. The resource factory reference name is the name of the client application's environment entry whose value contains the JNDI name of the data source.

res-type

The `res-type` element specifies the type of the data source. The type is specified by the Java interface or class expected to be implemented by the data source.

res-auth

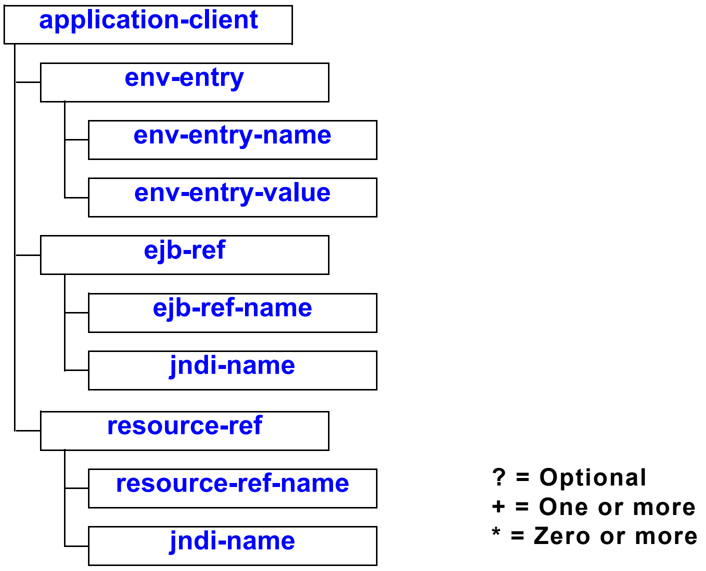
The `res-auth` element specifies whether the EJB code signs on programmatically to the resource manager, or whether the Container will sign on to the resource manager on behalf of the EJB. In the latter case, the Container uses information that is supplied by the Deployer. The res-auth element can have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

WebLogic Run-time Client Application Deployment Descriptor

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

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

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



application-client

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

env-entry

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

env-entry-name

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

Example:

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

env-entry-value

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

ejb-ref

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

ejb-ref-name

The `ejb-ref-name` element contains the name of an EJB reference. The EJB reference is an entry in the application client's environment. It is recommended that name is prefixed with `ejb/`.

Example:

```
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
```

jndi-name

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

resource-ref

The `resource-ref` element declares an application client's reference to an external resource. It contains the resource factory reference name, an indication of the resource factory type expected by the application client's code, and the type of authentication (bean or container).

Example:

```
<resource-ref>
  <res-ref-name>EmployeeAppDB</res-ref-name>
  <jndi-name>enterprise/databases/HR1984</jndi-name>
</resource-ref>
```

resource-ref-name

The `res-ref-name` element specifies the name of the resource factory reference name. The resource factory reference name is the name of the application client's environment entry whose value contains the JNDI name of the data source.

jndi-name

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

Index