BEA WebLogic Server

Developing WebLogic Server Applications

BEA WebLogic Server 6.0
Document Date: May 16, 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Developing WebLogic Server Applicatinos

Part Number	Document Date	Software Version
N/A	May 16, 2001	BEA WebLogic Server Version 6.0

Contents

About This Document

Audience x
e-docs Web Site x
How to Print the Document x
Related Information xi
Contact Us! xi
Documentation Conventions xii

1. Understanding WebLogic Server Applications

What Are WebLogic Server Application Components? 1-1
Web Application Components 1-2
Servlets 1-3
JavaServer Pages 1-3
Web Application Directory Structure 1-3
For More Information on Web Application Components 1-4
Enterprise JavaBean Components 1-4
EJB Overview 1-4
EJB Interfaces 1-5
EJBs and WebLogic Server 1-5
WebLogic Components 1-6
Enterprise Applications 1-7
Client Applications 1-7

2. Developing WebLogic Server Components

Creating WebLogic Server Applications: Main Steps 2-1
Creating Web Applications: Overview 2-3
Creating Enterprise Beans: Overview 2-4

Creating Enterprise Applications: Overview 2-5
Establishing a Development Environment 2-6
Software Tools 2-6
Source Code Editor or IDE 2-6
Java Compiler 2-6
Development WebLogic Server 2-7
Database System and JDBC Driver 2-7
Web Browser 2-8
Third-Party Software 2-8
WebGain VisualCafé Enterprise Edition 2-8
Informix Cloudscape 2-9
Sybase PowerJ 2-9
WebGain TopLink 2-9
KL Group JProbe 2-9
Versant Enterprise Container 2-10
eXcelon Javlin 2-10
Object Design ObjectStore 2-10
Preparing to Compile 2-10
Putting the Java Tools in Your Search Path 2-11
Setting the Classpath for Compiling 2-11
Setting Target Directories for Compiled Classes 2-12

3. Packaging and Deploying WebLogic Server Applications

Packaging Overview 3-1
JAR Files 3-2
XML Deployment Descriptors 3-3
Packaging Web Applications 3-4
Packaging Enterprise JavaBeans 3-5
Packaging Enterprise Applications 3-6
Resolving Class References Between Components 3-8
Classloader Overview 3-8
About Application Classloaders 3-9
Packaging Common Utilities and Third-Party Classes 3-10
Handling Interactions Between Startup Classes and Applications 3-10
Deploying Applications and Components 3-11

Using the Administration Console 3-12
Using the weblogic.deploy Command-Line Utility 3-13
Packaging and Deploying Client Applications 3-13
J2EE Client 3-14

4. Programming Topics

Logging Messages 4-1
Using Threads in WebLogic Server 4-4
Using JavaMail with WebLogic Server Applications 4-6
About JavaMail Configuration Files 4-6
Configuring JavaMail for WebLogic Server 4-7
Sending Messages with JavaMail 4-9
Reading Messages with JavaMail 4-10

5. Writing Web Application Deployment Descriptors

Overview of Web Application Deployment Descriptors 5-1
Writing the web.xml Deployment Descriptor 5-2
Main Steps to Create the web.xml File 5-2
Detailed Steps to Create the web.xml File 5-3
Writing the WebLogic-Specific Deployment Descriptor (weblogic.xml) 5-19
Main Steps to Create the weblogic.xml File 5-19
Detailed Steps to Create the weblogic.xml File 5-20

A. web.xml Deployment Descriptor Elements

icon Element A-2
display-name Element A-2
description Element A-3
context-param Element A-3
servlet Element A-3
icon Element A-5
init-param Element A-5
security-role-ref Element A-6
servlet-mapping Element A-6
session-config Element A-7
mime-mapping Element A-8
welcome-file-list Element A-9

error-page Element A-9
taglib Element A-10
resource-ref Element A-11
security-constraint Element A-12
web-resource-collection Element A-12
auth-constraint Element A-13
user-data-constraint Element A-14
login-config Element A-15
form-login-config Element A-15
security-role Element A-16
env-entry Element A-17
ejb-ref Element A-17

B. weblogic.xml Deployment Descriptor Elements

description Element B-1
weblogic-version Element B-2
security-role-assignment Element B-2
reference-descriptor Element B-3
resource-description Element B-3
ejb-reference-description Element B-3
session-descriptor Element B-4
Session Parameter Names and Values B-4
jsp-descriptor Element B-8
JSP Parameter Names and Values B-9

C. Client Application Deployment Descriptor Elements

application.xml Deployment Descriptor Elements C-1
application C-2
icon C-3
display-name C-3
description C-3
module C-3
security-role C-4
WebLogic Run-time Client Application Deployment Descriptor C-5
application-client C-6

env-entry* C-6
ejb-ref* C-7
resource-ref* C-7

About This Document

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

The document is organized as follows:

- Chapter 1, “Understanding WebLogic Server Applications,” describes components of WebLogic Server applications.
- Chapter 2, “Developing WebLogic Server Components,” describes the process for creating WebLogic Server components and helps Java programmers establish their programming environment.
- Chapter 3, “Packaging and Deploying WebLogic Server Applications,” describes how to bundle WebLogic Server components and applications in standard JAR files for distribution and deployment.
- Chapter 4, “Programming Topics,” covers general WebLogic Server application programming issues, such as logging messages and using threads.
- Chapter 5, “Writing Web Application Deployment Descriptors,” describes how to write the deployment descriptors that tell WebLogic Server how to deploy a Web application.
- Appendix A, “web.xml Deployment Descriptor Elements,” is a reference for the standard J2EE Web application deployment descriptor, `web.xml`.
- Appendix B, “weblogic.xml Deployment Descriptor Elements,” is a reference for the WebLogic-specific Web application deployment descriptor, `weblogic.xml`.
- Appendix C, “Client Application Deployment Descriptor Elements,” is a reference for the standard J2EE Client application deployment descriptor,

`application.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 or go directly to the WebLogic Server Product Documentation page at http://e-docs.bea.com/wls/docs60.

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 EJB*
- *Programming WebLogic HTTP Servlets*
- *Programming WebLogic JSP*
- *Programming WebLogic JDBC*

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

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR

Convention	Usage
`{ }`	A set of choices in a syntax line.

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

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

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

CHAPTER

1 Understanding WebLogic Server Applications

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

- What Are WebLogic Server Application Components?
- Web Application Components
- Enterprise JavaBean Components
- WebLogic Components
- Enterprise Applications
- Client Applications

What Are WebLogic Server Application Components?

BEA WebLogic Server™ applications can include the following components:

- Web components—HTML pages, servlets, JavaServer Pages, and related files

- EJB components—entity beans, session beans, and message-driven beans
- WebLogic components—startup and shutdown classes

Web designers, application developers, and application assemblers create components by using J2EE technologies such as JavaServer Pages, servlets, and Enterprise JavaBeans.

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 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.

An Enterprise Application, consisting of assembled components, is a JAR file with an `.ear` extension. An `.ear` file contains all of the `.jar` and `.war` component archive files for an application and an XML descriptor that describes the bundled components.

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

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

Web Application Components

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

- Servlets, JSP pages, and their helper classes.
- HTML/XML pages with supporting files such as images and multimedia files.
- 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.

Servlets

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

JavaServer Pages

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

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

Web Application Directory Structure

Web application components are assembled in a directory in order to stage the `.war` file for the `jar` command. HTML pages, JSP pages, and the non-Java class files they reference are accessed beginning in the top level of the staging directory.

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

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

For More Information on Web Application Components

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

- Programming WebLogic Servlets at http://e-docs.bea.com/wls/docs60/servlet/index.html
- Programming WebLogic JSP at http://e-docs.bea.com/wls/docs60/jsp/index.html
- Writing JSP Extensions at http://e-docs.bea.com/wls/docs60/taglib/index.html

For help deploying Web Applications, see the following sections of this document:

- Chapter 5, "Writing Web Application Deployment Descriptors."
- Appendix A, "web.xml Deployment Descriptor Elements."
- Appendix B, "weblogic.xml Deployment Descriptor Elements."

Enterprise JavaBean Components

Enterprise JavaBeans (EJBs) beans are server-side Java components 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 represent a single client within WebLogic Server. They 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 shared 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 on the JMS Queue, the message-driven bean assigns an instance of itself from a pool to process the message. Message-driven beans are not associated with any client. They simply handle messages as they arrive. A JMS ServerSessionPool provides a similar capability, but without the advantages of running in the EJB container.

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

EJB Interfaces

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

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

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

EJBs and WebLogic Server

J2EE cleanly separates the development and deployment roles to ensure that components are portable between EJB servers that support the EJB specification. Deploying an enterprise bean in WebLogic Server requires running the WebLogic EJB compiler, `weblogic.ejbc`, to generate the stub and skeleton classes that allow an enterprise bean to be executed remotely.

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

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

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

For help creating and deploying Enterprise JavaBeans, see Programming WebLogic Enterprise JavaBeans at http://e-docs.bea.com/wls/docs60/ejb/index.html.

WebLogic Components

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

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

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

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

Enterprise Applications

An Enterprise Archive (`.ear`) file contains the Web archives and EJB archives that constitute a J2EE application. The `META-INF/application.xml` deployment descriptor contains an entry for each Web and EJB module, and additional entries to describe security roles and application resources such as databases.

You use the Administration Console or the `weblogic.deploy` command line utility to deploy an `.ear` file on one or more WebLogic Servers in the domain managed by the Administration Server.

Client Applications

Client-side applications written in Java have access to WebLogic Server services via RMI. Client applications range from simple command line utilities that use standard I/O to highly interactive GUI applications built using the Java Swing/AWT classes.

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

To execute a WebLogic Server Java client, the client computer needs the `weblogic_sp.jar` file, 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` and `weblogic_sp.jar` files.

WebLogic Server also supports J2EE client applications, packaged in a JAR file with a standard XML 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 and Deploying Client Applications" on page 3-13 for more about J2EE client applications.

2 Developing WebLogic Server Components

The following sections describe how to create WebLogic Server components and set up a development envioronment:

- Creating WebLogic Server Applications: Main Steps
- Establishing a Development Environment
- Preparing to Compile

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 WebLogic Server Applications: Main Steps

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

Here are the main steps:

1. Create Web and EJB components for your application.

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

2. Create deployment descriptors.

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

3. Create component archive.

   Component archives are JAR files containing all of the files that make up the component, including deployment descriptors.

4. Create application deployment descriptor.

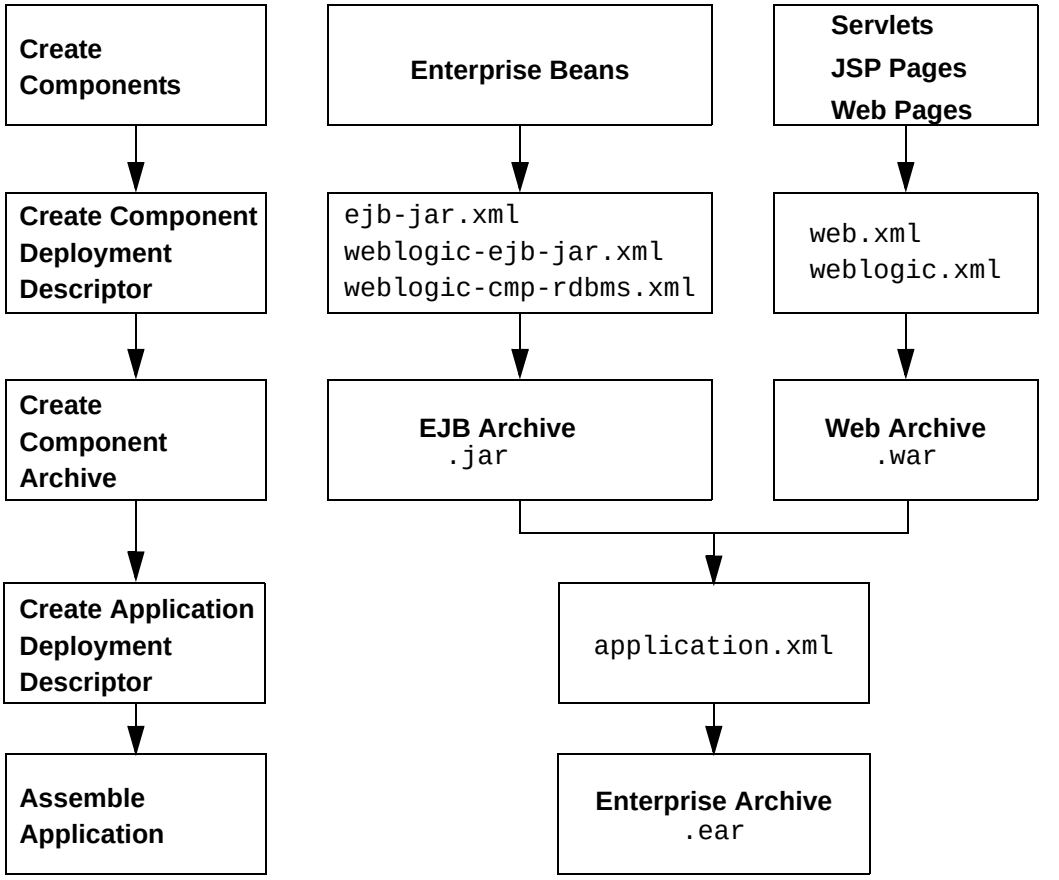
   The application deployement descriptor, `application.xml`, lists individual components that are assembled together in an application.

5. Assemble application.

   Component archives are packaged with the application deployment descriptor in an application JAR 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 JAR file.

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

Figure 2-1 Creating Enterprise Applications



Creating Web Applications: Overview

Web designers create HTML pages and JavaServer Pages to create the Web interface for an application. Java programmers create Servlets and the JSP taglibs referenced in JavaServer Pages (JSPs).

JSP pages, HTML pages, and multimedia files referenced by the pages are saved in the top level of the Web application staging directory. Compiled Servlet classes, taglibs, and, if desired, Servlets compiled from JSP pages are stored under a `WEB-INF` directory in the staging directory.

Two deployment descriptors are created in the `WEB-INF` directory: `web.xml` and `weblogic.xml`. 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. See Chapter 5, "Writing Web Application Deployment Descriptors," for instructions on creating these deployment descriptors.

When the Web application components are all in place in the staging directory, you create the Web archive with a command such as the following, executed in the staging directory:

```
jar cvf myWebApp.war *
```

The `.war` file is ready to be deployed, or it can be added to an Enterprise archive and deployed as part of an application.

Creating Enterprise Beans: Overview

A Java programmer creates an enterprise bean by writing three classes, in accordance with the EJB specification:

- An EJB home interface
- A remote interface for the bean
- An implementation class for the bean

Message-driven beans require only an implementation class. The interfaces and implementation classes are compiled into a staging directory for the bean.

Deployment descriptors are created in a `META-INF` directory in the top level of the staging directory:

- `ejb-jar.xml` describes the enterprise bean 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 bean to tables in a database. This file can must have a different name for each CMP bean packaged in a JAR file. The name of the file is specified in the bean's entry in the `weblogic-ejb.jar` file.

See "Programming WebLogic Enterprise JavaBeans" at http://e-docs.bea.com/wls/docs60/ejb/index.html for help creating EJB deployment descriptors.

After the EJB classes are compiled, you run the `weblogic.ejbc` EJB compiler to generate the stub and skeleton classes into the staging directory. Then you create the EJB archive by executing a `jar` command like the following in the staging directory:

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

The EJB `.jar` file can be deployed as is, or packaged in an Enterprise Archive (`.ear`) file and deployed with an application.

Creating Enterprise Applications: Overview

When you have assembled all of the Web archives and EJB archives for your application, you can bundle them together in an Enterprise Archive (`.ear`) file so that you can deploy all of the dependent components together.

Copy the `.war` and EJB `.jar` files into a staging directory and then create a `META-INF/application.xml` deployment descriptor for the application. The `application.xml` file contains a descriptor for each component in the application, using a DTD supplied by Sun Microsystems.

Create the Enterprise Archive by executing a `jar` command such as the following in the staging directory:

```
jar cvf myApp.ear *
```

Use the Administration Console or the `weblogic.deploy` command line utility to deploy the application.

Establishing a Development Environment

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

Software Tools

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

Source Code Editor or IDE

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

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

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

Java Compiler

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

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

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

Occasionally, a compiler generates optimized code that does not behave well in all Java Virtual Machines (JVMs). When debugging 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 of your target JVMs before deploying.

Development WebLogic Server

Never deploy untested code on a WebLogic Server that is serving production applications, so you will need a development WebLogic Server in your environment. You can run a development WebLogic Server on the same computer you edit and compile on, or you can use one deployed somewhere on the network.

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

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

Database System and JDBC Driver

Nearly all WebLogic Server applications require a database system. You can use any DBMS that you can access with a standard JDBC driver, but services such as WebLogic JMS require a supported JDBC driver for Oracle, Sybase, Informix,

Microsoft SQL Server, IBM DB2, or Cloudscape. Refer to the Platform Support Web page at http://e-docs.bea.com/wls/platforms/index.html to find out about supported database systems and JDBC drivers.

JDBC connection pools offer such significant performance advantages that you should only rarely consider writing an application that uses a two-tier JDBC driver directly. On a WebLogic 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 SSL? Test alternative security settings in the browser so that you can tell your users what choices you support.

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

Third-Party Software

This section describes several third-party software products that can enhance your WebLogic Server development environment.

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

WebGain VisualCafé Enterprise Edition

VisualCafé Enterprise Edition builds on proven Java development technology and delivers a complete Java Integrated Development Environment (IDE) for the heterogeneous enterprise. Enterprise Edition inherits powerful productivity features

and database functionality from the VisualCafé family while introducing unique new server-side development and distributed debugging capabilities to minimize the challenges of enterprise Java application development. VisualCafé Enterprise Edition is the most mature and open Java development environment backed by extensive industry wide support.

Informix Cloudscape

Cloudscape is the first 100% pure Java SQL database management system. Cloudscape is designed to be embedded in client or server applications as a local data manager. It implements SQL-92 with extensions for Java that enable the developer to create a column of type Java class, and to write stored procedures in Java for execution inside the DBMS.

Sybase PowerJ

PowerJ provides a true end-to-end solution for building sophisticated Internet applications, exploiting the benefits of HTML, Java clients, and Java server-side components.

WebGain TopLink

TOPLink for BEA WebLogic is a powerful tool for building EJB applications. TOPLink is seamlessly integrated with BEA WebLogic Server. With TOPLink for BEA WebLogic Server, you can build components for application servers that run on Java, while significantly cutting application development time and expense.

KL Group JProbe

JProbe is a performance-tuning toolkit. With other performance-tuning toolkits, developers working on specific application servers must use command line prompts. Under these circumstances, setting up the session typically involves time-consuming manual configuration, and usually more than a fair share of trial and error. The JProbe 2.5 Server Launch Pad eliminates much of this effort, making server-side tuning fast and easy. JProbe supports many popular Web and application servers, including WebLogic Server.

Versant Enterprise Container

VERSANT Enterprise Container (VEC) is the integration between a VERSANT ODBMS and an EJB-compliant application server. VEC supports WebLogic Server. VERSANT Enterprise Container is an EJB-compliant container that plugs directly into the application server, allowing transparent persistence for your entity beans. VERSANT Enterprise Container is targeted at applications that have complex object models and require high-performance access to persistent data.

eXcelon Javlin

EJB is the standard for building scalable eBusiness applications, from eCommerce, supply-chain, and customer-relationship management, to enterprise information and application portals, and others.

EJB-compliant application servers facilitate the creation, distribution, and integration of eBusiness Java components in the middle tier—but they do not do the same for data. In order to meet or exceed time to market and performance goals, you need a middle-tier data manager like Javlin to complement your EJB server.

Object Design ObjectStore

ObjectStore is an ideal data management solution for developers creating dynamic, reliable, high-performance applications for telecommunications, packaged software, the Internet and other distributed computing environments.

ObjectStore combines best-of-breed, object data management with Java and C++, and ActiveX to enable the development and delivery of high-speed, complex Web transactions, dynamic content, and network management applications.

Preparing to Compile

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

- Have the Java compiler in your search path

- Set your classpath so that the Java compiler can find all of the dependent classes
- Specify the output directories for the compiled classes

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

Putting the Java Tools in Your Search Path

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

For example, if the JDK is installed in `/usr/local/java/java130` on your UNIX file system, use a command such as the following to add `javac` to your path in a Bourne shell or shell script:

```
PATH=/usr/local/java/java130/bin:$PATH; export PATH
```

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

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

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

Setting the Classpath for Compiling

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

- The `lib/tools.jar` file in the JDK directory, or other standard Java classes required by the Java Development Kit you use.
- The `weblogic_sp.jar` file distributed with a WebLogic Server service pack, if you have one.

  This `jar` file should precede `weblogic.jar` in your classpath so that the service pack classes are found before any classes they supersede in `weblogic.jar`.
- Classes for third party Java tools or services your programs import.
- Other application classes referenced by the programs you are compiling.

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

Setting Target Directories for Compiled Classes

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

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

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

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

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

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

`APPLICATIONS`
: The `applications` directory for the examples domain. Unlike the others, this variable is not used to specify a target for the Java compiler. It is used as a convenient reference to the `applications` directory in copy commands that move files from source directories into the `applications` directory. For example, if you have `.html`, `.jsp`, and image files in your source tree, you can use the variable in a copy command to install them in your development server.

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

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

If you do not use an IDE, consider writing a make file, shell script, or command file to compile and package your components and applications. Set the variables in the build script so that you can rebuild components by typing a single command.

CHAPTER

3 Packaging and Deploying WebLogic Server Applications

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

- Packaging Overview
- Packaging Web Applications
- Packaging Enterprise JavaBeans
- Packaging Enterprise Applications
- Resolving Class References Between Components
- Deploying Applications and Components
- Packaging and Deploying Client Applications

Packaging Overview

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

J2EE includes deployment specifications for Web applications, EJB modules, Enterprise applications, and Client applications. 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, Web pages and supporting files, XML-formatted deployment descriptors, and JAR files containing other components.

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

JAR Files

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

JAR files are convenient for packaging components and applications for distribution. They are easier to copy, they use up fewer file handles than an exploded directory, and they can save disk space with file compression. If your Administration Server manages a domain with multiple WebLogic Servers, you can only deploy JAR files, because the Administration Console does not copy expanded directories to managed servers.

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

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

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

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

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

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

XML Deployment Descriptors

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

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

Table 3-1 J2EE and WebLogic Deployment Descriptors

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

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

If you receive a J2EE-compliant JAR file from a developer, it already contains J2EE-defined deployment descriptors. To deploy the JAR file on WebLogic Server, you must extract the contents of the JAR file into a directory, add the required WebLogic-specific deployment descriptors and any generated container classes, and then create a new JAR file containing the old and new files.

Packaging Web Applications

To stage and package a Web application:

1. Create a temporary staging directory.
2. Copy all of your HTML files, JSP files, images, and any other files that these Web pages reference into the staging directory, maintaining the directory structure for referenced files. For example, if an HTML file has a tag such as `<img src="images/pic.gif">`, the `pic.gif` file must be in the `images` subdirectory beneath the HTML file.
3. Create `META-INF` and `WEB-INF/classes` subdirectories in the staging directory to hold deployment descriptors and compiled Java classes.
4. Copy or compile any servlet classes and helper classes into the `WEB-INF/classes` subdirectory.
5. Copy the home and remote interface classes for enterprise beans used by the servlets into the `WEB-INF/classes` subdirectory.

   Note: See “Classloader Overview” on page 3-8 to understand how the WebLogic Server class-loading mechanism affects EJB references from servlets within the same application.
6. Copy JSP tag libraries into the `WEB-INF` subdirectory. (Tag libraries may be installed in a subdirectory beneath `WEB-INF`; the path to the `.tld` file is coded in the `.jsp` file.)

7. Create `web.xml` and `weblogic.xml` deployment descriptors in the `WEB-INF` subdirectory.

   Note: See "Writing Web Application Deployment Descriptors" on page 5-1 for help creating deployment descriptors for Web applications.

8. Bundle the staging directory into a `.war` file by executing a `jar` command such as the following:

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

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

Packaging Enterprise JavaBeans

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

To stage and package an enterprise bean:

1. Create a temporary staging directory.
2. Compile or copy the bean's Java classes into the staging directory.
3. Create a `META-INF` subdirectory in the staging directory.
4. Create an `ejb-jar.xml` deployment descriptor in the `META-INF` subdirectory and add entries for the bean.
5. Create a `weblogic-ejb-jar.xml` deployment descriptor in the `META-INF` subdirectory and add entries for the bean.
6. 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 EJB*" at http://e-docs.bea.com/wls/docs60/ejb/index.html for help compiling enterprise beans and creating EJB deployment descriptors.

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

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

   This command creates a `jar` file that you can deploy on a WebLogic Server or package in an application JAR file.

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

Enterprise beans require *container classes*, classes the WebLogic EJB compiler generates to allow the bean to deploy in a WebLogic Server. The WebLogic EJB compiler reads the deployment descriptors in the EJB JAR file to determine how to generate the classes. You can run the WebLogic EJB compiler on the JAR file before you deploy the beans, or you can let WebLogic Server run the compiler for you at deployment time. See *Programming WebLogic EJB* at http://e-docs.bea.com/wls/docs60/ejb/index.html for help with the WebLogic EJB compiler.

Packaging Enterprise Applications

An Enterprise archive contains EJB and Web modules that are part of a related application. The EJB and Web modules are bundled together in another JAR file with an `.ear` extension.

The `META-INF` subdirectory in an `.ear` file contains an `application.xml` deployment descriptor, which identifies the modules packaged in the `.ear` file. You can find the DTD for the `application.xml` file at `http://java.sun.com/j2ee/dtds/application_1_2.dtd`. No WebLogic-specific deployment descriptor is needed for an enterprise archive.

Here is the `application.xml` file from the Pet Store example:

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

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

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

To stage and package an Enterprise application:

1. Create a temporary staging directory.
2. Copy the Web archives (`.war` files) and EJB archives (`.jar` files) into the staging directory.
3. Create a `META-INF` subdirectory in the staging directory.
4. Create the `application.xml` deployment descriptor in the `META-INF` subdirectory.
5. Create the Enterprise Archive (`.ear` file) for the application, using a `jar` command such as:

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

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

Resolving Class References Between Components

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

Classloader Overview

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

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

The search protocol also clarifies how duplicate classes are handled in Java. Classes located in the Java system classpath always have precedence over any class with the same name in a child classloader's classpath. Because of this, you should avoid placing

application classes in the Java system classpath before you start WebLogic Server. The classloader created at startup time cannot be destroyed, so any classes it contains cannot be redeployed without restarting WebLogic Server.

About Application Classloaders

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

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

- Package the servlets and JSPs in a `.war` file
- Package the enterprise beans in an EJB `.jar` file
- Package the `.war` and `.jar` files in an `.ear` file
- Deploy the `.ear` file

Although you could deploy the `.war` and `.jar` files separately, deploying them together in an `.ear` file produces a classloader arrangement that allows the servlets and JSPs to find the EJB classes. If you deploy the `.war` and `.ejb` files separately, WebLogic Server creates sibling classloaders for them. 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.

Packaging Common Utilities and Third-Party Classes

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

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.

Handling Interactions Between Startup Classes and Applications

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

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

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

The Pet Store application has a startup class that demonstrates one method a startup class can use to wait for applications to finish deploying. The `com.bea.estore.startup.StartBrowser` startup class displays the initial URL to access the Pet Store application, and on Windows it also launches the browser with the URL. `StartBrowser` executes a `while` loop until applications have deployed and the server begins accepting connection requests.

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

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

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

If the system fails to create a socket, or if the BEA-supplied `beaexec.exe` utility returns an error, the class sleeps for 500 milliseconds before repeating the loop. If a startup class needs to create an EJB instance, it could use a similar technique by looping until the EJB create method succeeds.

Deploying Applications and Components

You can deploy an EJB JAR, Web application, or Enterprise application by using the Administration Console or the `weblogic.deploy` command-line utility. You can also use either method to undeploy or redeploy an updated application.

The `.jar`, `.ear`, or `.war` file must be correctly structured and contain all of the necessary deployment descriptors described in previous sections.

Note: If you have a single WebLogic Server, you can deploy applications and components by copying them to the server's `applications` subdirectory. In this case, files do not need to be packaged in JAR files. The ability to refresh individual files in the `applications` directory is useful for testing during development. However, using JAR files is recommended for production applications.

Using the Administration Console

To deploy an application using the Administration Console:

1. Start the Administration Console.
2. In the left pane, expand Deployments.
3. Under Deployments, click Applications.
4. In the right pane, click Browse, and find the `.ear`, `.jar`, or `.war` file containing the component or application you want to install.
5. Click Upload.

   This copies the file to the Administration Server's `applications` directory.
6. Expand the new application under the Applications node to reveal the components.
7. For each of the components in the application, click the component name in the left pane, then complete the information on the Configuration and Targets tabs in the right pane. Consult the online help to find details about the values on these tabs.
8. Click on the application name under the Applications node, and check the Deployed check box in the right pane.
9. Click Apply.

Depending on your choices, you may need to restart WebLogic Server. The Administration Console displays a restart message in the right pane.

Using the weblogic.deploy Command-Line Utility

The `weblogic.deploy` command-line utility allows you to deploy, undeploy, update, and list components on an Administration Server—tasks you can accomplish interactively using the Administration Console. The `weblogic.deploy` command line utility can be used in scripts, which is especially useful during development.

See the documentation for `weblogic.deploy` in the *Administration Guide* for syntax and usage for this command line-utility.

Packaging and Deploying Client Applications

WebLogic Server applications written in Java run in a JVM on a client machine. 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. This usually means distributing the `weblogic_sp.jar` and `weblogic.jar` files to the client and adding them to the client's classpath.

You may also want to package a Java Runtime Environment (JRE) with a Java client application.

You can stage a client application by copying all of the files required on the client into a directory and bundling the directory up in a `.zip` file, or a `.jar` file if you know that clients have already installed a Java environment with the `jar` utility.

The top level of the client application directory can have a batch file or script to start the application. Make a `classes` subdirectory to hold Java classes and `.jar` files, and add them to the client's classpath in the startup script.

J2EE Client

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. The `.jar` file contains the Java classes that execute in the client JVM and deployment descriptors that describe Enterprise JavaBeans and other WebLogic resources used by the client.

A standard deployment descriptor from Sun is used for J2EE clients and a supplemental deployment descriptor contains additional WebLogic-specific deployment information.

See "Client Application Deployment Descriptor Elements" on page C-1 for help with these deployment descriptors.

On the client, the `weblogic.ClientDeployer` utility starts a J2EE client application on the client machine. This 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 an `.ear` extension, that contains one or more Client application `.jar` files.

For example:

```
java appclient.ClientDeployer app.ear client
```

In this example, the `app.ear` file is a JAR file that contains a J2EE client packaged in the `client.jar` JAR file.

CHAPTER

4 Programming Topics

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

- Logging Messages
- Using Threads in WebLogic Server
- Using JavaMail with WebLogic Server Applications

Logging Messages

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

This section describes how to use the `NonCatalogLogger` class. See the *Internationalization Guide* at http://e-docs.bea.com/wls/docs60/i18n/index.html for details on using the internationalization interface.

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

Table 4-1 Log Message Format

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

Table 4-1 Log Message Format

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

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

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

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

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

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

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

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

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

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

Using Threads in WebLogic Server

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

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

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

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

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

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

To avoid undesireable 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 occuring in production.

Using JavaMail with WebLogic Server Applications

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

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

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

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

About JavaMail Configuration Files

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

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

Configuring JavaMail for WebLogic Server

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

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

Table 4-2 Mail Session Properties Field

Property	Description	Default
`mail.store.protocol`	The protocol to use to retrieve email. Example: `mail.store.protocol=imap`	The bundled JavaMail library has support for IMAP.
`mail.transport.protocol`	The protocol to use to send email. Example: `mail.transport.protocol=smtp`	The bundled JavaMail library has support for SMTP.

Table 4-2 Mail Session Properties Field

Property	Description	Default
`mail.host`	The name of the mail host machine. Example: `mail.host=mailserver`	The default is the local machine.
`mail.user`	The name of the default user for retrieving email. Example: `mail.user=postmaster`	The default is the value of the `user.name` Java system property.

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

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

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

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

Sending Messages with JavaMail

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

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

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

2. Look up the Mail Session in JNDI:

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

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

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

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

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

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

5. Send the message.

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

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

Reading Messages with JavaMail

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

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

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

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

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

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

2. Look up the Mail Session in JNDI:

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

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

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

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

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

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

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

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

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

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

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

5 Writing Web Application Deployment Descriptors

The following sections describe how to writeWeb Application deployment descriptors:

- Overview of Web Application Deployment Descriptors
- Writing the web.xml Deployment Descriptor
- Writing the WebLogic-Specific Deployment Descriptor (weblogic.xml)

Overview of Web Application Deployment Descriptors

Deploying Web Applications requires you to create two deployment descriptors for each Web Application. These deployment descriptors define components and operating parameters for a Web Application. Deployment descriptors are standard text

files, formatted using XML notation and are packaged within the Web Application. For more information on Web Applications, see Deploying and Configuring Web Applications at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html.

The first deployment descriptor, `web.xml` is defined by the Servlet 2.2 specification from Sun Microsystems. This deployment descriptor can be used to deploy a Web Application on any J2EE-compliant application server.

The second deployment descriptor, `weblogic.xml`, defines deployment properties that are specific to a Web Application running on WebLogic Server.

Writing the web.xml Deployment Descriptor

This section describes the steps to create the `web.xml` deployment descriptor. Depending on the components in your Web Application, you may not need to include all of the elements listed here to configure and deploy your Web Application.

The elements in the `web.xml` *file must be entered in the order they are presented in this document.*

Main Steps to Create the web.xml File

Step 1: Create a deployment descriptor file on page 5-3
Step 2: Create the header on page 5-3
Step 3: Create the main body of the web.xml file on page 5-4
Step 4: Define deployment-time attributes on page 5-4
Step 5: Define context parameters on page 5-5
Step 6: Deploy servlets on page 5-6
Step 7: Map a servlet to a URL on page 5-8
Step 8: Define the session timeout value on page 5-9
Step 9: Define welcome pages on page 5-9
Step 10: Define error pages on page 5-10
Step 11: Define MIME mapping on page 5-10
Step 12: Define a JSP tag library descriptor on page 5-11
Step 13: Reference external resources on page 5-12
Step 14: Set up security constraints on page 5-12

Step 15: Set up login authentication on page 5-14
Step 16: Define security roles on page 5-16
Step 17: Set environment entries on page 5-16
Step 18: Reference Enterprise JavaBean (EJB) resources on page 5-17

If you have installed the WebLogic Server samples and examples, you can look at the `web.xml` and `weblogic.xml` files in the Pet Store sample to see a working example of Web Application deployment descriptors. These files are located in the `/samples/PetStore/source/dd/war/WEB-INF` directory of your WebLogic Server distribution.

Detailed Steps to Create the web.xml File

Step 1: Create a deployment descriptor file

Name the file `web.xml` and place it under the `WEB-INF` directory of the Web Application. Use any text editor.

Step 2: Create the header

This text must be the first line of the file:

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
 "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

The header refers to the location and version of the Document Type Descriptor (DTD) file for the deployment descriptor. Although this header references an external URL at `java.sun.com`, WebLogic Server contains its own copy of the DTD file, so your host server need not have access to the Internet. However, you must still include this `<!DOCTYPE...>` element in your `web.xml` file, and have it reference the external URL because the version of the DTD contained in this element is used to identify the version of this deployment descriptor.

Step 3: Create the main body of the web.xml file

Wrap all of your entries within a pair of opening and closing `<web-app>` tags.

`<web-app>` All elements describing this Web Application go within the `<web-app>` element. **`</web-app>`**	This tag should be the final tag in the `web.xml` file

In XML, properties are defined by surrounding a property name or value with opening and closing tags as shown above. The opening tag, the body (the property name or value), and the closing tag are collectively called an *element*. Some elements do not use the surrounding tags, but instead use a single tag that contains attributes called an *empty-tag*. Elements contained within other elements are indented in this text for clarity. Indenting is not necessary in an XML file.

The body of the `<web-app>` element itself contains additional elements that determine how the Web Application will run on WebLogic Server. The order of the tag elements within the file *must follow the order reflected in this document*. This ordering is defined in the Document Type Descriptor (DTD) file. For more information, refer to the DTD, available on the Sun Microsystems Web site at http://java.sun.com/j2ee/dtds/web-app_2_2.dtd.

Step 4: Define deployment-time attributes

These tags provide information for the deployment tools or the application server resource management tools. These values are not used by WebLogic Server in this release.

`<small-icon>` `iconfile.gif(jpg)` **`</small-icon>`**	(Optional)
`<large-icon>` `iconfile.gif(jpg)` **`</large-icon>`**	(Optional)
`<display-name>` `application-name` **`</display-name>`**	(Optional)

<description> descriptive-text **</description>**	(Optional)
<distributable>	(Optional)

Step 5: Define context parameters

The `context-param` element declares a Web Application's servlet context initialization parameters. These can be parameters that you define that will be available throughout your Web Application. You set each `context-param` within a single `context-param` element, using `<param-name>` and `<param-value>` elements. You can access these parameters in your code using the `javax.servlet.ServletContext.getInitParameter()` and `javax.servlet.ServletContext.getInitParameterNames()` methods.

Precompiling JSPs

You can use the `context-param` element to specify that WebLogic Server precompile JSPs on start up. For more information, see Precompiling JSPs at `http://e-docs.bea.com/wls/docs60/jsp/reference.html#precompile`.

<context-param>	For more information, see context-param Element on page A-3
<param-name> user-defined param name **</param-name>**	(Required)
<param-value> user-defined value **</param-value>** **<description** text description **</description>**	(Required)
<context-param>	

Step 6: Deploy servlets

In this step, you give the servlet a name, specify the class file or JSP used to implement its behavior, and set other servlet-specific properties. List each of the servlets in your Web Application within separate `<servlet>...</servlet>` elements. After you create entries for all of your servlets, you must include elements that map the servlet to a URL pattern. These mapping elements are described in "Step 7: Map a servlet to a URL" on page 5-8.

Use the following elements to declare a servlet:

<servlet>	For more information, see "servlet Element" on page A-3
<servlet-name> name **</servlet-name>**	(Required)
<servlet-class> package.name.MyClass **</servlet-class>** -or- **<jsp-file>** /foo/bar/myFile.jsp **</jsp-file>**	(Required)
<init-param>	For more information, see "init-param Element" on page A-5
<param-name> name **</param-name>**	*(Required)*
<param-value> value **</param-value>**	*(Required)*
<description> ...text... **</description>** **</init-param>**	*(Optional)*
<load-on-startup> loadOrder **</load-on-startup>**	*(Optional)*

<security-role-ref>	*(Optional).* For more information, see "security-role-ref Element" on page A-6
<description> ...text... **</description>**	*(Optional)*
<role-name> rolename **</role-name>**	*(Required)*
<role-link> rolelink **</role-link>**	*(Required)*
</security-role-ref>	
<small-icon> iconfile **</small-icon>**	*Not Used.(Optional)*
<large-icon> iconfile **</large-icon>**	*(Optional)*
<display-name> Servlet Name **</display-name>**	*(Optional)*
<description> ...text... **</description>**	*(Optional)*
</servlet>	

Here is an example of a servlet element that includes an initialization parameter.

```
<servlet>
   <init-param>
       <param-name>feedbackEmail</param-name>
       <param-value>feedback123@beasys.com</param-value>
       <description>
         The email for web-site feedback.
       </description>
     </init-param>
   </servlet>
```

Step 7: Map a servlet to a URL

Once you declare your servlet or JSP using a `<servlet>` element, map it to one or more URL patterns to make it a public HTTP resource. For each mapping, use a `<servlet-mapping>` element.

`<servlet-mapping>`	For more information, see servlet-mapping Element on page A-6
`<servlet-name>` `name` **`</servlet-name>`**	*(Required)*
`<url-pattern>` `pattern` **`</url-pattern>`**	*(Required)*
`</servlet-mapping>`	

Here is an example of a `<servlet-mapping>` for the `<servlet>` declaration example used earlier:

```
<servlet-mapping>
  <servlet-name>LoginServlet</servlet-name>
  <url-pattern>/login</url-pattern>
</servlet-mapping>
```

Step 8: Define the session timeout value

<session-config>	(Optional)
<session-timeout> minutes **</session-timeout>**	For more information, see "session-config Element" on page A-7
</session-config>	

Step 9: Define welcome pages

<welcome-file-list>	(Welcome pages are Optional.) For more information, see "welcome-file-list Element" on page A-9
<welcome-file> myWelcomeFile.jsp **</welcome-file>** **<welcome-file>** myWelcomeFile.html **</welcome-file>**	See also Welcome Pages at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#welcome_pages And How WebLogic Server Resolves HTTP Requests at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#resolve_http_req
</welcome-file-list>	

Step 10: Define error pages

```
<error-page>
```

(Optional) Define a customized page to respond to errors

For more information, see "error-page Element" on page A-9

And How WebLogic Server Resolves HTTP Requests at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#resolve_http_req

```
    <error-code>
       HTTP error code
    </error-code>
```

-or-

```
    <exception-type>
       Java exception class
    </exception-type>
    <location>URL</location>
</error-page>
```

Step 11: Define MIME mapping

```
<mime-mapping>
```

(Optional)

Define MIME types

For more information, see "mime-mapping Element" on page A-8

```
      <extension>
         ext
      </extension>
      <mime-type>
         mime type
      </mime-type>
</mime-mapping>
```

Step 12: Define a JSP tag library descriptor

<taglib>	*(Optional)* Identify JSP tag libraries For more information, see "taglib Element" on page A-10
<taglib-uri> string_pattern **</taglib-uri>**	*(Required)*
<taglib-location> filename **</taglib-location>**	*(Required)*
</taglib>	

The following is an example of a taglib directive used in a JSP:

```
<%@ taglib uri="string_pattern" prefix="taglib" %>
```

For more details, see the Programming WebLogic JSP Tag Extensions at http://e-docs.bea.com/wls/docs60/taglib/index.html.

Step 13: Reference external resources

<resource-ref>	*(Optional)* For more information, see "resource-ref Element" on page A-11.
<res-ref-name> name **</res-ref-name>**	*(Required)*
<res-type> Java class **</res-type>**	*(Required)*
<res-auth> CONTAINER | SERVLET **</res-auth>**	*(Required)*
</resource-ref>	

Step 14: Set up security constraints

A Web Application that uses security requires the user to log in in order to access its resources. The user's credentials are verified against a security realm, and once authorized, the user will have access only to specified resources within the Web Application.

Security in a Web Application is configured using three elements:

- The **<login-config>** element specifies how the user is prompted to login and the location of the security realm. If this element is present, the user must be authenticated in order to access any resource that is constrained by a <security-constraint> defined in the Web Application.
- A **<security-constraint>** is used to define the access privileges to a collection of resources via their URL mapping.
- A **<security-role>** element represents a group or principal in the realm. This security role name is used in the <security-constraint> element and can be linked to an alternative role name used in servlet code via the <security-role-ref> element.

<security-constraint>	*(Optional)* For more information, see "security-constraint Element" on page A-12
<web-resource-collection>	*(Required)* For more information, see "web-resource-collection Element" on page A-12
<web-resource-name> name **</web-resource-name>**	*(Required)*
<description> ...text... **</description>**	*(Optional)*
<url-pattern> pattern **</url-pattern>**	*(Optional)*
<http-method> GET | POST **</http-method>**	*(Optional)*
</web-resource-collection>	
<auth-constraint>	*(Optional)* For more information, see "auth-constraint Element" on page A-13
<role-name> group | principal **</role-name>**	*(Optional)*
</auth-constraint>	

<user-data-constraint>	*(Optional)* For more information, see "user-data-constraint Element" on page A-14
<description>...text...**</description>**	*(Optional)*
<transport-guarantee> NONE INTEGRAL or CONFIDENTIAL **</transport-guarantee>**	*(Required)*
</user-data-constraint>	
</security-constraint>	

Step 15: Set up login authentication

<login-config>	*(Optional)* For more information, see "login-config Element" on page A-15
<auth-method> BASIC,FORM, or CLIENT-CERT **</auth-method>**	*(Optional)* Specifies the method used to authenticate the user

<realm-name> realmname **</realm-name>**	*(Optional)* For more information, see Specifying a Security Realm at http://e-docs.bea.com/wls/docs60/adminguide/cnfgsec.html#cnfgsec004.
<form-login-config>	*(Optional)* For more information, see "form-login-config Element" on page A-15 Use this element if you configure the `<auth-method>` to `FORM`
<form-login-page> URI **</form-login-page>**	*(Required)*
<form-error-page> URI **</form-error-page>** **</form-login-config>**	*(Required)*
</login-config>	

Step 16: Define security roles

<security-role>	*(Optional)* For more information, see "security-role Element" on page A-16
<description> ...text... **</description>**	*(Optional)*
<role-name> rolename **</role-name>**	*(Required)*
</security-role>	

Step 17: Set environment entries

<env-entry>	*(Optional)* For more information, see "env-entry Element" on page A-17
<description> ...text... **</description>**	*(Optional)*
<env-entry-name> name **</env-entry-name>**	*(Required)*
<env-entry-value> value **</env-entry-value>**	*(Required)*
<env-entry-type> type **</env-entry-type>**	*(Required)*
</env-entry>	

Step 18: Reference Enterprise JavaBean (EJB) resources

<ejb-ref>	*Optional)* For more information, see "ejb-ref Element" on page A-17
<description> ...text... **</description>**	*(Optional)*
<ejb-ref-name> **name** **</ejb-ref-name>**	*(Required)*
<ejb-ref-type> Java type **</ejb-ref-type>**	*(Required)*
<home> **mycom.ejb.AccountHome** **</home>**	*(Required)*
<remote> mycom.ejb.Account **</remote>**	*(Required)*
<ejb-link> ejb.name **</ejb-link>**	*(Optional)*
</ejb-ref>	

Listing 5-1 Sample web.xml with Servlet Mapping, Welcome file, and Error Page

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//
DTD Web Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

<!- The following servlet element defines a servlet called servletA.
 The Java class of this servlet is servlets.servletA ->
  <servlet>
    <servlet-name>servletA</servlet-name>
```

```
    <servlet-class>servlets.servletA</servlet-class>
  </servlet>

<!- The following servlet element defines another servlet called
 servletB. The Java class of this servlet is servlets.servletB ->
  <servlet>
    <servlet-name>servletB</servlet-name>
    <servlet-class>servlets.servletB</servlet-class>
  </servlet>

<!- The following servlet-mapping maps the servlet called servletA
 (see the servlet element) to a url-pattern of "blue".
 The url-pattern is used when requesting this servlet, for example:
 http://host:port/myWebApp/blue. ->
  <servlet-mapping>
    <servlet-name>servletA</servlet-name>
    <url-pattern>blue</url-pattern>
  </servlet-mapping>

<!- The following servlet-mapping maps the servlet called servletB
 (see the servlet element) to a url-pattern of "yellow".
 The url-pattern is used when requesting this servlet, for example:
 http://host:port/myWebApp/yellow. ->
  <servlet-mapping>
    <servlet-name>servletB</servlet-name>
    <url-pattern>yellow</url-pattern>
  </servlet-mapping>

<!-The following welcome-file-list specifies a welcome-file.
 Welcome files are discussed elsewhere in this document->
  <welcome-file-list>
    <welcome-file>hello.html</welcome-file>
  </welcome-file-list>

<!-The following error-page element specifies a page that is served
 in place of the standard HTTP error response pages, in this case
 HTTP error 404.->
  <error-page>
    <error-code>404</error-code>
    <location>/error.jsp</location>
  </error-page>

</web-app>
```

Writing the WebLogic-Specific Deployment Descriptor (weblogic.xml)

The `weblogic.xml` file contains WebLogic-specific attributes for a Web Application. You define the following attributes in this file: HTTP session parameters, HTTP cookie parameters, JSP parameters, resource references, and security role assignments.

If you define external resources such as DataSources, EJBs, or a Security realm in the `web.xml` deployment descriptor, you can use any descriptive name to define the resource. To access the resource, you then map this resource name to the actual name of the resource in the JNDI tree using a file called `weblogic.xml`. Place this file in the `WEB-INF` directory of your Web Application.

If you have installed the WebLogic Server samples and examples, you can look at the `web.xml` and `weblogic.xml` files in the Pet Store sample to see a working example of Web application deployment descriptors. These files are located in the `/samples/PetStore/source/dd/war/WEB-INF` directory of your WebLogic Server distribution.

The ordering of the tag elements within the `weblogic.xml` file must follow the ordering specified in this document.

Main Steps to Create the weblogic.xml File

Step 1: Begin the weblogic.xml file with a DOCTYPE header on page 5-20
Step 2: Map security role names to a security realm on page 5-21
Step 3 Reference resources on page 5-21
Step 4: Define session parameters on page 5-23
Step 5: Define JSP parameter on page 5-23

Detailed Steps to Create the weblogic.xml File

Step 1: Begin the weblogic.xml file with a DOCTYPE header

This header refers to the location and version of the DTD file for the deployment descriptor. Although this header references an external URL at `www.beasys.com`, WebLogic Server has its own copy of the DTD file, so your host server need not have access to the Internet. However, you must still include this `DOCTYPE` element in your `weblogic.xml` file, and have it reference the external URL since the version of the DTD is used to identify the version of this deployment descriptor.

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA
 Systems, Inc.//DTD Web Application 6.0//EN"
 "http://www.bea.com/servers/wls600/dtd/
weblogic-web-jar.dtd">
<weblogic-web-app>
     <description>
        Text description of the Web App
     </description>
     <weblogic-version>
     </weblogic-version>
```

This element is not used by WebLogic Server

Step 2: Map security role names to a security realm

<security-role-assignment>	
<role-name> name **</role-name>**	*(Required)* For more information, see "security-role-assignment Element" on page B-2
<principal-name> name **</principal-name>**	*(Required)*
</security-role-assignment>	

If you need to define multiple roles, define each additional pair of `<role-name>` and `<principal-name>` tags within separate `<security-role-assignment>` elements.

Step 3 Reference resources

In this step you map resources used in your Web Application to the JNDI tree. When you define an `<ejb-ref-name>` or a `<res-ref-name>` in the `web.xml` deployment descriptor, you also reference those names in `weblogic.xml` and map them to an actual JNDI name that is available in WebLogic Server. In the following example, a Data Source is referenced in a servlet with the name `myDataSource`. `myDataSource` is then referenced in `web.xml` and its data type defined. Finally, in the `weblogic.xml` file, `myDataSource` is mapped to the JNDI name `accountDataSource`, which is available in the JNDI tree. The JNDI name must match the name of an object bound in the JNDI tree. Objects can be bound to the JNDI tree programatically or by configuring them in the Administration Console. For more information, see Programming WebLogic JNDI at `{DOCROOTS}/jndi/index.html`.

Servlet code:

```
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup
                          ("myDataSource");
```

`web.xml` entries:

```
<resource-ref>
. . .
   <res-ref-name>myDataSource</res-ref-name>
```

```
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>CONTAINER</res-auth>
. . .
</resource-ref>
```

weblogic.xml entries:

```
<resource-description>
   <res-ref-name>myDataSource</res-ref-name>
   <jndi-name>accountDataSource</jndi-name>
</security-role-ref>
```

A similar pattern is used to map EJBs to the JNDI tree, but uses the `<ejb-ref-name>` element of the `<ejb-reference-description>` element in place of the `<res-ref-name>` element of the `<resource-description>` element.

<reference-descriptor>	For more information, see "reference-descriptor Element" on page B-3
<resource-description>	For more information, see "resource-description Element" on page B-3
<res-ref-name> name **</res-ref-name>**	*(Required)*
<jndi-name> JNDI name of resource **</jndi-name>**	*(Required)*
</resource-description>	
<ejb-reference-description>	
<ejb-ref-name> name **</ejb-ref-name>**	*(Required)* For more information, see "ejb-reference-description Element" on page B-3
<jndi-name> JNDI name of EJB **</jndi-name>**	*(Required)*

```
    </ejb-reference-description>
</reference-descriptor>
```

Step 4: Define session parameters

You define HTTP session parameters for this Web Application inside of `<session-param>` tags, which are nested in side `<session-descriptor>` tags. For each `<session-param>` you need to supply a `<param-name>...</param-name>` element that names the parameter being defined and a `<param-value>...</param-value>` element that provides the value of the parameter. For a list of HTTP session parameters and details on setting them, see "session-descriptor Element" on page B-4 .

```
<session-descriptor>                                  For more information, see "session-descriptor Element" on page B-4
    <session-param>
        <param-name>
           session param name
        </param-name>
        <param-value>
            my value
        </param-value>
    </session-param>
</session-descriptor>
```

Step 5: Define JSP parameter

You define JSP configuration parameters for this Web Application inside of `<jsp-param>` tags, which are nested in side `<jsp-descriptor>` tags. For each `<jsp-param>` you need to supply a `<param-name>...</param-name>` element that

names the parameter being defined and a `<param-value>...</param-value>` element that provides the value of the parameter. For a list of JSP parameters and details on setting them, see "jsp-descriptor Element" on page B-8.

```
<jsp-descriptor>                                    For more information,
                                                    see "jsp-descriptor
                                                    Element" on page B-8
     <jsp-param>
          <param-name>
            jsp param name
          </param-name>
          <param-value>
             my value
          </param-value>
     </jsp-param>
</jsp-descriptor>
```

A web.xml Deployment Descriptor Elements

This following sections describe the deployment descriptor elements defined in the `web.xml` file. The root element for `web.xml` is `<web-app>`. The following elements are defined within the `<web-app>` element:

- "icon Element" on page A-2
- "display-name Element" on page A-2
- "description Element" on page A-3
- "context-param Element" on page A-3
- "servlet Element" on page A-3
- "servlet-mapping Element" on page A-6
- "session-config Element" on page A-7
- "mime-mapping Element" on page A-8
- "welcome-file-list Element" on page A-9
- "error-page Element" on page A-9
- "taglib Element" on page A-10
- "resource-ref Element" on page A-11
- "security-constraint Element" on page A-12
- "login-config Element" on page A-15
- "env-entry Element" on page A-17

■ "ejb-ref Element" on page A-17

icon Element

The `icon` element specifies the location within the Web Application for a small and large image used to represent the Web Application in a GUI tool. (The servlet element also has an element called the icon element, used to supply an icon to represent a servlet in a GUI tool.)

This element is not currently used by WebLogic Server.

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

Element	Required/ Optional	Description
`<small-icon>`	Optional	Specifies the location for a small (16x16 pixel) `.gif` or `.jpg` image used to represent the Web 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 Web Application in a GUI tool. Currently, this element is not used by WebLogic Server.
`<display-name>`	Optional	Currently, this element is not used by WebLogic Server.
`<description>`	Optional	Currently, this element is not used by WebLogic Server.
`<distributable>`	Optional	Currently, this element is not used by WebLogic Server.

display-name Element

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

description Element

The optional description element provides descriptive text about the Web Application.

context-param Element

The optional `context-param` element declares a Web Application's servlet context initialization parameters. You set each context-param within a single context-param element, using <param-name> and <param-value> elements. You can access these parameters in your code using the `javax.servlet.ServletContext.getInitParameter()` and `javax.servlet.ServletContext.getInitParameterNames()` methods.

The following table describes the elements you can define within a `context-param` element.

Element	Required/ Optional	Description
`<param-name>`	Required	The name of a parameter.
`<param-value>`	Required	The value of a parameter.
`<description>`	Required	A text description of a parameter.

servlet Element

The `servlet` element contains the declarative data of a servlet.

If a `jsp-file` is specified and the load-on-startup element is present, then the JSP should be precompiled and loaded.

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

Element	Required/ Optional	Description
`<icon>`	Optional	Specifies the location within the Web Application for a small and large image used to represent the servlet in a GUI tool. Contains a small-icon and large-icon element. Currently, this element is not used by WebLogic Server.
`<servlet-name>`	Required	Defines the canonical name of the servlet, used to reference the servlet definition elsewhere in the deployment descriptor.
`<display-name>`	Optional	A short name intended to be displayed by GUI tools.
`<description>`	Optional	A text description of the servlet.
`<servlet-class>`	Required (or use `<jsp-file>`	The fully-qualified class name of the servlet. You may use only one of either the `<servlet-class>` tags or `<jsp-file>` tags in your servlet body.
`<jsp-file>`	Required (or use `<servlet-class>`	The full path to a JSP file within the Web Application, relative to the Web Application root directory. You may use only one of either the `<servlet-class>` tags or `<jsp-file>` tags in your servlet body.
`<init-param>`	Optional	Contains a name/value pair as an initialization parameter of the servlet. Use a separate set of `<init-param>` tags for each parameter.
`<load-on-startup>`	Optional	WebLogic Server initializes this servlet when WebLogic Server starts up. The optional contents of this element must be a positive integer indicating the order in which the servlet should be loaded. Lower integers are loaded before higher integers. If no value is specified, or if the value specified is not a positive integer, WebLogic Server can load the servlet in any order in the startup sequence.
`<security-role-ref>`	Optional	Used to link a security role name defined by `<security-role>` to an alternative role name that is hard coded in the servlet logic. This extra layer of abstraction allows the servlet to be configured at deployment without changing servlet code.

icon Element

This is an element within the "servlet Element" on page A-3.

The `icon` element specifies the location within the Web Application for small and large images used to represent the servlet in a GUI tool.

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

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

init-param Element

This is an element within the "servlet Element" on page A-3.

The optional `init-param` element contains a name/value pair as an initialization parameter of the servlet. Use a separate set of `init-param` tags for each parameter.

You can access these parameters with the `javax.servlet.ServletConfig.getInitParameter()` method.

The following table describes the elements you can define within a `init-param` element.

Element	Required/ Optional	Description
`<param-name>`	Required	Defines the name of this parameter.
`<param-value>`	Required	Defines a `String` value for this parameter.

Element	Required/ Optional	Description
<description>	Optional	Text description of the initialization parameter.

security-role-ref Element

This is an element within the "servlet Element" on page A-3.

The security-role-ref element links a security role name defined by <security-role> to an alternative role name that is hard-coded in the servlet logic. This extra layer of abstraction allows the servlet to be configured at deployment without changing servlet code.

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

Element	Required/ Optional	Description
<description>	Optional	Text description of the role.
<role-name>	Required	Defines the name of the security role or principal that is used in the servlet code.
<role-link>	Required	Defines the name of the security role that is defined in a <security-role> element later in the deployment descriptor.

servlet-mapping Element

The servlet-mapping element defines a mapping between a servlet and a URL pattern.

The following table describes the elements you can define within a `servlet-mapping` element.

Element	Required/ Optional	Description
`<servlet-name>`	Required	The name of the servlet to which you are mapping a URL pattern. This name corresponds to the name you assigned a servlet in a `<servlet>` declaration tag.

`<url-pattern>`	Required	Describes a pattern used to resolve URLs. The portion of the URL after the `http://host:port` + `WebAppName` is compared to the `<url-pattern>` by WebLogic Server. If the patterns match, the servlet mapped in this element will be called.
		Example patterns:
		`/soda/grape/*`
		`/foo/*`
		`/contents`
		`*.foo`
		The URL must follow the rules specified in Section 10 of the Servlet 2.2 Specification.
		For additional examples of servlet mapping, see Servlet Mapping http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#servlet-mapping.

session-config Element

The `session-config` element defines the session parameters for this Web Application.

The following table describes the element you can define within a `session-config` element.

Element	Required/ Optional	Description
`<session-timeout>`	Optional	The number of minutes after which sessions in this Web Application expire. The value set in this element overrides the value set in the `TimeoutSecs` parameter of the `<session-descriptor>` element in the WebLogic-specific deployment descriptor `weblogic.xml`, unless one of the special values listed here is entered. Default value: -2 Maximum value: Integer.MAX_VALUE ÷ 60 Special values: ■ -2 = Use the value set by `TimeoutSecs` in `<session-descriptor>` element of `weblogic.xml` ■ -1 = Sessions do not timeout. The value set in `<session-descriptor>` element of `weblogic.xml` is ignored. For more information, see "session-descriptor Element" on page B-4.

mime-mapping Element

The `mime-mapping` element defines a mapping between an extension and a mime type.

The following table describes the elements you can define within a `mime-mapping` element.

Element	Required/ Optional	Description
`<extension>`	Required	A string describing an extension, for example: `txt`.
`<mime-type>`	Required	A string describing the defined mime type, for example: `text/plain`.

welcome-file-list Element

The optional `welcome-file-list` element contains an ordered list of `welcome-file` elements.

When the URL request is a directory name, WebLogic Server serves the first file specified in this element. If that file is not found, the server then tries the next file in the list.

For more information, see Welcome Files at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#welcome_pages and How WebLogic Server Resolves HTTP Requests at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#resolve_http_req.

The following table describes the element you can define within a `welcome-file-list` element.

Element	Required/ Optional	Description
`<welcome-file>`	Optional	File name to use as a default welcome file, such as index.html

error-page Element

The optional `error-page` element specifies a mapping between an error code or exception type to the path of a resource in the Web Application.

When an error occurs—while WebLogic Server is responding to an HTTP request, or as a result of a Java exception—WebLogic Server returns an HTML page that displays either the HTTP error code or a page containing the Java error message. You can define your own HTML page to be displayed in place of these default error pages or in response to a Java exception.

For more information, see Customizing HTTP Error Responses at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#error_pages and How WebLogic Server Resolves HTTP Requests at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#resolve_http_req.

The following table describes the elements you can define within an `error-page` element.

Element	Required/ Optional	Description
`<error-code>`	Optional	A valid HTTP error code, for example `404`.
`<exception-type>`	Optional	A fully-qualified class name of a Java exception type, for example `java.lang`.
`<location>`	Required	The location of the resource to display in response to the error. For example `/myErrorPg.html`

taglib Element

The optional `taglib` element describes a JSP tag library.

This element associates the location of a JSP Tag Library Descriptor (TLD) with a URI pattern. Although you can specify a TLD in your JSP that is relative to the WEB-INF directory, you can also use the `<taglib>` tag to configure the TLD when deploying your Web Application. Use a separate element for each TLD.

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

Element	Required/ Optional	Description
`<taglib-location>`	Required	Gives the file name of the tag library descriptor relative to the root of the Web Application. It is good idea to store the tag library descriptor file under the `WEB-INF` directory so it is not publicly available over an HTTP request.

Element	Required/ Optional	Description
<taglib-uri>	Required	Describes a URI, relative to the location of the web.xml document, identifying a Tag Library used in the Web Application. If the URI matches the URI string used in the taglib directive on the JSP page, this taglib is used.

resource-ref Element

The optional resource-ref element defines a reference lookup name to an external resource. This allows the servlet code to look up a resource by a "virtual" name that is mapped to the actual location at deployment time.

Use a separate <resource-ref> element to define each external resource name. The external resource name is mapped to the actual location name of the resource at deployment time in the WebLogic-specific deployment descriptor weblogic.xml.

The following table describes the elements you can define within a resource-ref element.

Element	Required/ Optional	Description
<description>	Optional	A text description.
<res-ref-name>	Required	The name of the resource used in the JNDI tree. Servlets in the Web Application use this name to look up a reference to the resource.
<res-type>	Required	The Java type of the resource that corresponds to the reference name. Use the full package name of the Java type.
<res-auth>	Required	Used to control the resource sign on for security. If set to SERVLET, indicates that the application component code performs resource sign on programmatically. If set to CONTAINER WebLogic Server uses the security context established with the login-config element. See "login-config Element" on page A-15.

security-constraint Element

The `security-constraint` element defines the access privileges to a collection of resources via their URL mapping.

For more information, see Configuring Security in Web Applications at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#configure-security.

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

Element	Required/ Optional	Description
`<web-resource-collection>`	Required	Defines the components of the Web Application that this security constraint is applied to.
`<auth-constraint>`	Optional	Defines which groups or principals have access to the collection of web resources defined in this security constraint. See also "auth-constraint Element" on page A-13.
`<user-data-constraint>`	Optional	Defines how the client should communicate with the server. See also "user-data-constraint Element" on page A-14.

web-resource-collection Element

Each `<security-constraint>` element must have one or more `<web-resource-collection>` elements. These define the area of the Web Application that this security constraint is applied to.

This is an element within the "security-constraint Element" on page A-12.

The required `web-resource-collection` element define the area of the Web Application that this security constraint is applied to.

The following table describes the elements you can define within a `web-resource-collection` element.

Element	Required/ Optional	Description
`<web-resource-name>`	Required	The name of this Web resource collection.
`<description>`	Optional	A text description of this security constraint.
`<url-pattern>`	Optional	Use one or more of these elements to declare which URL patterns this security constraint applies to. If you do not use at least one of these elements, this <web-resource-collection> is ignored by WebLogic Server.
`<http-method>`	Optional	Use one or more of these elements to declare which HTTP methods (`GET | POST |...`) are subject to the authorization constraint. If you omit this element, the default behavior is to apply the security constraint to all HTTP methods.

auth-constraint Element

This is an element within the "security-constraint Element" on page A-12.

The optional `auth-constraint` element defines which groups or principals have access to the collection of Web resources defined in this security constraint.

The following table describes the elements you can define within an `auth-constraint` element.

Element	Required/ Optional	Description
`<description>`	Optional	A text description of this security constraint.
`<role-name>`	Optional	Defines which security roles can access resources defined in this security-constraint. Security role names are mapped to principals using the `security-role-ref Element`. See "security-role-ref Element" on page A-6.

user-data-constraint Element

This is an element within the "security-constraint Element" on page A-12.

The `user-data-constraint` element defines how the client should communicate with the server.

The following table describes the elements you may define within a `user-data-constraint` element.

Element	Required/ Optional	Description
`<description>`	Optional	A text description.
`<transport-guarantee>`	Required	Specifies that the communication between client and server. WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the `INTEGRAL` or `CONFIDENTIAL` constraint. Range of values: ■ `NONE`—the application does not require any transport guarantees. ■ `INTEGRAL`—the application requires that the data sent between the client and server be sent in such a way that it cannot be changed in transit. ■ `CONFIDENTIAL`—the application requires that the data be transmitted in a fashion that prevents other entities from observing the contents of the transmission.

login-config Element

The optional `login-config` element configures how the user is authenticated, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism.

If this element is present, the user must be authenticated in order to access any resource that is constrained by a `<security-constraint>` defined in the Web Application. Once authenticated, the user can be authorized to access other resources with access privileges.

The following table describes the elements you can define within a `login-config` element.

Element	Required/ Optional	Description
`<auth-method>`	Optional	Specifies the method used to authenticate the user. Possible values: `BASIC` - uses browser authentication `FORM` - uses a user-written HTML form `CLIENT-CERT`
`<realm-name>`	Optional	The name of the realm that is referenced to authenticate the user credentials. If omitted, the WebLogic realm is used by default. For more information, see Specifying a Security Realm at http://e-docs.bea.com/wls/docs60/adminguide/cnfgsec.html#cnfgsec004.
`<form-login-config>`	Optional	Use this element if you configure the `<auth-method>` to FORM. See "form-login-config Element" on page A-15.

form-login-config Element

This is an element within the "login-config Element" on page A-15.

Use the form-login-config element if you configure the `<auth-method>` to `FORM`.

.

Element	Required/ Optional	Description
`<form-login-page>`	Required	The URI of a Web resource relative to the document root, used to authenticate the user. This can be an HTML page, JSP, or HTTP servlet, and must return an HTML page containing a FORM that conforms to a specific naming convention. For more information, see Setting Up Authentication for Web Applications at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#webapp-auth
`<form-error-page>`	Required	The URI of a Web resource relative to the document root, sent to the user in response to a failed authentication login.

security-role Element

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

Element	Required/ Optional	Description
`<description>`	Optional	A text description of this security role.
`<role-name>`	Required	The role name. The name you use here must have a corresponding entry in the WebLogic-specific deployment descriptor, `weblogic.xml`, which maps roles to principals in the security realm. For more information, see "security-role-assignment Element" on page B-2.

env-entry Element

The optional `env-entry` element declares an environment entry for an application. Use a separate element for each environment entry.

The following table describes the elements you can define within a `env-entry` element.

Element	Required/ Optional	Description
`<description>`	Optional	A textual description.
`<env-entry-name>`	Required	The name of the environment entry.
`<env-entry-value>`	Required	The value of the environment entry.

`<env-entry-type>`	Required	The type of the environment entry.
		Can be set to one of the following Java types:
		`java.lang.Boolean`
		`java.lang.String`
		`java.lang.Integer`
		`java.lang.Double`
		`java.lang.Float`

ejb-ref Element

The optional `ejb-ref` element defines a reference to an EJB resource. This reference is mapped to the actual location of the EJB at deployment time by defining the mapping in the WebLogic-specific deployment descriptor file, `weblogic.xml`. Use a separate `<ejb-ref>` element to define each reference EJB name.

The following table describes the elements you can define within a `ejb-ref` element.

Element	Required/ Optional	Description
<description>	Optional	A text description of the reference.
<ejb-ref-name>	Required	The name of the EJB used in the Web Application. This name is mapped to the JNDI Tree in the WebLogic-specific deployment descriptor `weblogic.xml`. For more information, see "ejb-reference-description Element" on page B-3.
`<ejb-ref-type>`	Required	The expected Java class type of the referenced EJB.
`<home>`	Required	The fully qualified class name of the EJB home interface.
`<remote>`	Required	The fully qualified class name of the EJB remote interface.
`<ejb-link>`	Optional	The `<ejb-name>` of an EJB in an encompassing J2EE application package.

B weblogic.xml Deployment Descriptor Elements

This following sections describe the deployment descriptor elements defined in the `weblogic.xml` file. The root element for `weblogic.xml` is `<weblogic-web-app>`. The following elements are defined within the `<weblogic-web-app>` element:

- "description Element" on page B-1
- "weblogic-version Element" on page B-2
- "security-role-assignment Element" on page B-2
- "reference-descriptor Element" on page B-3
- "session-descriptor Element" on page B-4
- "jsp-descriptor Element" on page B-8

You can also access the Document Type Descriptor (DTD) for `weblogic.xml` at http://www.bea.com/servers/wls600/dtd/weblogic-web-jar.dtd.

description Element

The description element is a text description of the Web Application.

weblogic-version Element

The `weblogic-version` element indicates the version of WebLogic Server on which this Web Application is intended to be deployed. This element is informational only and is not used by WebLogic Server.

security-role-assignment Element

The `security-role-assignment` element declares a mapping between a security role and one or more principals in the realm, as shown in the following example.

```
<security-role-assignment>
  <role-name>PayrollAdmin</role-name>
  <principal-name>Tanya</principal-name>
  <principal-name>Fred</principal-name>
  <principal-name>system</principal-name>
</security-role-assignment>
```

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

Element	Required Optional	Description
`<role-name>`	Required	Specifies the name of a security role.
`<principal-name>`	Required	Specifies the name of a principal that is defined in the security realm. You can use multiple `<principal-name>` elements to map principals to a role. For more information on security realms, see the *Programming WebLogic Security* at http://e-docs.bea.com/wls/docs60/security/index.html.

reference-descriptor Element

The `reference-descriptor` element maps the JNDI name of a server resource to a name used in the Web Application. The `reference-description` element contains two elements: The `resource-description` element maps a resource, for example, a DataSource, to its JNDI name. The `ejb-reference` element maps an EJB to its JNDI name.

resource-description Element

The following table describes the elements you can define within a `resource-description` element.

Element	Required/ Optional	Description
<res-ref-name>	Required	Specifies the name of a resource reference.
<jndi-name>	Required	Specifies a JNDI name for the resource.

ejb-reference-description Element

The following table describes the elements you can define within a `ejb-reference-description` element.

Element	Required/ Optional	Description
<ejb-ref-name>	Required	Specifies the name of an EJB reference used in your Web Application.
<jndi-name>	Required	Specifies a JNDI name for the reference.

session-descriptor Element

The `session-descriptor` element defines parameters for HTTP sessions, as shown in the following example:

```
<session-descriptor>
  <session-param>
     <param-name>
       CookieDomain
     </param-name>
     <param-value>
       myCookieDomain
     </param-value>
  </session-param>
</session-descriptor>
```

Session Parameter Names and Values

The following table describes the valid session parameter names and values you can define within a `session-param` element:

Parameter Name	Default Value	Parameter Value
`CookieDomain`	Null	Identifies the server to which the browser sends cookie information when the browser makes a request. For example, setting the CookieDomain to `.mydomain.com` returns cookies to any server in the `*.mydomain.com` domain. The domain name must have at least two components; setting a name to `*.com` or `*.net` is invalid. If unset, this parameter defaults to the server that issued the cookie.
`CookieComment`	Weblogic Server Session Tracking Cookie	Specifies the comment that identifies the session tracking cookie in the cookie file. If unset, this parameter defaults to `WebLogic Session Tracking Cookie`. You may provide a more specific name for your application.

Parameter Name	Default Value	Parameter Value
`CookieMaxAgeSecs`	-1	Sets the life span of the session cookie, in seconds, after which it expires on the client. If the value is 0, the cookie expires immediately. The maximum value is MAX_VALUE, where the cookie lasts forever. If set to `-1`, the cookie expires when the user exits the browser. For more information about cookies, see Setting up Session Management at http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#session-management.
`CookieName`	`JSESSIONID`	Defines the session cookie name. Defaults to `JSESSIONID` if unset. You may set this to a more specific name for your application.
`CookiePath`	Null	Specifies the pathname to which the browser sends cookies. If unset, this parameter defaults to `/` (slash), where the browser sends cookies to all URLs served by WebLogic Server. You may set the path to a narrower mapping, to limit the request URLs to which the browser sends cookies.
`CookiesEnabled`	True	Use of session cookies is enabled by default and is recommended, but you can disable them by setting this property to `false`. You might turn this option off to test URL re-writing (see http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html#urlrewriting) on your site.
`InvalidationIntervalSecs`	`60`	Sets the time, in seconds, that WebLogic Server waits between doing house-cleaning checks for timed-out and invalid sessions, and deleting the old sessions and freeing up memory. Use this parameter to tune WebLogic Server for best performance on high traffic sites. The minimum value is every second (1). The maximum value is once a week (604,800 seconds). If unset, the parameter defaults to 60 seconds.

Parameter Name	Default Value	Parameter Value
`PersistentStoreDir`	`session_db`	If you have set `PersistentStoreType` to `file`, this parameter sets the directory path where WebLogic Server will store the sessions. The directory path is either relative to the temp directory or an absolute path. The temp directory is either a generated directory under the `WEB-INF` directory of the Web Application, or a directory specified by the context-param `javax.servlet.context.tmpdir`. Ensure that you have enough disk space to store the *number of valid sessions* multiplied by the *size of each session*. You can find the size of a session by looking at the files created in the `PersistentStoreDir`. You can make file-persistent sessions clusterable by making this directory a shared directory among different servers. You must create this directory manually.
`PersistentStorePool`	None	Specifies the name of a JDBC connection pool to be used for persistence storage. For more details on setting up a database connection pool, see Managing JDBC Connectivity at http://e-docs.bea.com/wls/docs60/adminguide/jdbc.html .
`PersistentStoreType`	`memory`	Sets the persistent store method to one of the following options: ■ `memory`—disables persistent session storage ■ `file`—uses file-based persistence (See also `PersistentStoreDir`, above) ■ `jdbc`—uses a database to store persistent sessions. (see also `PersistentStorePool`, above) ■ `replicated`—same as `memory`, but session data is replicated across the clustered servers
`SwapIntervalSecs`	10	Sets the time, in seconds, that WebLogic Server waits between purging the least recently-used sessions from the cache to the persistent store, when the `cacheEntries` limit has been reached. If unset, this property defaults to 10 seconds; minimum is 1 second, and maximum is 604800 (1 week).

Parameter Name	Default Value	Parameter Value
`IDLength`	52	Sets the size of the session ID. The minimum value is 8 bytes and the maximum value is `Integer.MAX_VALUE`. If you are writing a WAP application, you must use URL rewriting because the WAP protocol does not support cookies. Also, some WAP devices have a 128-character limit on URL length (including parameters), which limits the amount of data that can be transmitted using URL re-writing. To allow more space for parameters, use this parameter to limit the size of the session ID that is randomly generated by WebLogic Server
`CacheSize`	1024	The number of sessions that may be active at one time.
`TimeoutSecs`	3600	Sets the time, in seconds, that WebLogic Server waits before timing out a session, where x is the number of seconds between a session's activity. Minimum value is 1, default is 3600, and maximum value is integer MAX_VALUE. On busy sites, you can tune your application by adjusting the timeout of sessions. While you want to give a browser client every opportunity to finish a session, you do not want to tie up the server needlessly if the user has left the site or otherwise abandoned the session. This parameter can be overridden by the `session-timeout` element (defined in minutes) in `web.xml`. For more information, see "`session-config Element`" on page A-7.
`JDBConnectionTimeoutSecs`	120	Sets the time, in seconds, that WebLogic Server waits before timing out a JDBC connection, where x is the number of seconds between.
`URLRewritingEnabled`	true	Enables URL rewriting, which encodes the session ID into the URL and provides session tracking if cookies are disabled in the browser.

Parameter Name	Default Value	Parameter Value
ConsoleMainAttribute		If you enable Session Monitoring in the WebLogic Server Administration Console, set this parameter to the name of the session parameter you will use to identify each session that is monitored. For more information, see Monitoring a WebLogic Domain at http://e-docs.bea.com/wls/docs60/adminguide/monitoring.html.

jsp-descriptor Element

The `jsp-descriptor` element defines parameter names and values for servlet JSPs, as shown in the following example.

```
<jsp-descriptor>
  <jsp-param>
    <param-name>
     FOO
    </param-name>
    <param-value>
     BAR
    </param-value>
  </jsp-param>
</jsp-descriptor>
```

JSP Parameter Names and Values

The following table describes the parameter names and values you can define within a `jsp-param` element.

Parameter Name	Default Value	Parameter Value

`compileCommand`	`javac`, or the Java compiler defined for a server under the configuration /tuning tab of the WebLogic Server Administration Console	Specifies the full pathname of the standard Java compiler used to compile the generated JSP servlets. For example, to use the standard Java compiler, specify its location on your system as shown below:
		`<param-value>`
		`  /jdk130/bin/javac.exe`
		`</param-value>`
		You can also specify that WebLogic Server precompile JSPs on start up. For more information, see Precompiling JSPs at `http://e-docs.bea.com/wls/docs60/jsp/reference.html#precompile`.

`compileFlags`	`None`	Passes one or more command-line flags to the compiler. Enclose multiple flags in quotes, separated by a space. For example:
		`java weblogic.jspc`
		`-compileFlags "-g -v" myFile.jsp`

`compilerclass`	`None`	Name of a Java compiler that is executed in WebLogic Servers's virtual machine. (Used in place of an executable compiler such as `javac` or `sj`.)

`encoding`	Default encoding of your platform	Specifies the default character set used in the JSP page. Use standard Java character set names (see http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.htm).
		If unset, this parameter defaults to the encoding for your platform.
		A JSP page directive (included in the JSP code) overrides this setting. For example:
		`<%@ page contentType="text/html;`
		`charset=custom-encoding"%>`

`keepgenerated`	`false`	Saves the Java files that are generated as an intermediary step in the JSP compilation process. Unless this parameter is set to `true`, the intermediate Java files are deleted after they are compiled.

Parameter Name	Default Value	Parameter Value
noTryBlocks	false	If a JSP file has numerous or deeply nested custom JSP tags and you receive a java.lang.VerifyError exception when compiling, use this flag to allow the JSPs to compile correctly.
packagePrefix	jsp_servlet	Specifies the package into which all JSP pages are compiled.
pageCheckSeconds	1	Sets the interval, in seconds, at which WebLogic Server checks to see if JSP files have changed and need recompiling. Dependencies are also checked and recursively reloaded if changed. If set to 0, pages are checked on every request. If set to -1, page checking and recompiling is disabled.
verbose	true	When set to true, debugging information is printed out to the browser, the command prompt, and WebLogic Server log file.
workingDir	internally generated directory	The name of a directory where WebLogic Server saves the generated Java and compiled class files for a JSP.

C Client Application Deployment Descriptor Elements

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

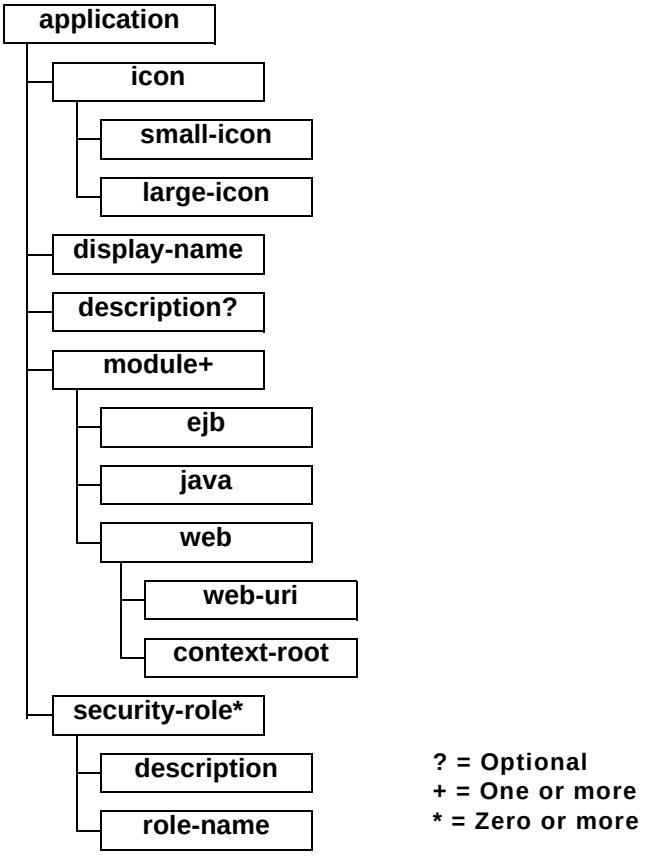
- application.xml Deployment Descriptor Elements
- WebLogic Run-time Client Application Deployment Descriptor

application.xml Deployment Descriptor Elements

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

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

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



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

application

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

icon

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

small-icon

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

large-icon

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

display-name

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

description

The optional description element provides descriptive text about the application.

module

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

ejb

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

Example:

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

java

Defines a client application module in the application file.

Example:

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

web

Defines a Web application module in the application file. The `web` element contains a `web-uri` element and, optionally, a `context-root` element.

web-uri

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

context-root

Optional. Specifies a context root for the Web application.

Example:

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

security-role

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

description

Optional. Text description of the security role.

role-name

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

Example:

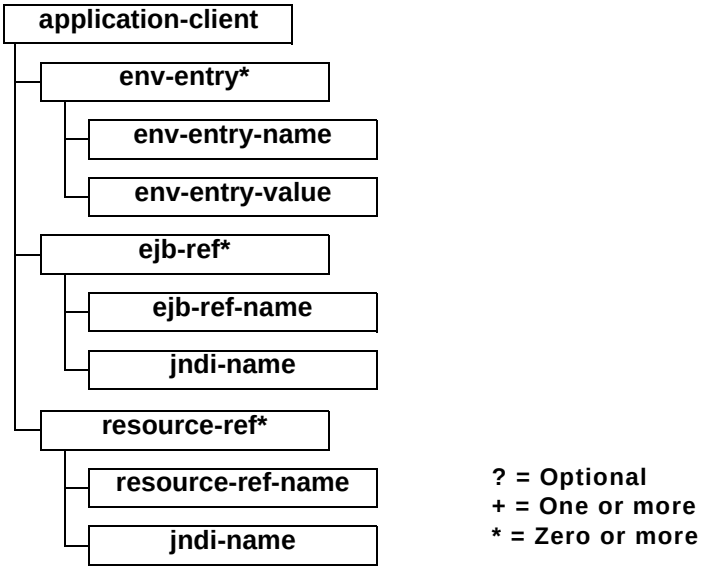
```
<security-role>
  <description>the gold customer role</description>
  <role-name>gold_customer</role-name>
</security-role>
<security-role>
  <description>the customer role</description>
  <role-name>customer</role-name>
</security-role>
```

WebLogic 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 runtime deployment descriptor is in the file named `c:/applications/ClientMain.runtime.xml`.

The following diagram shows the structure of the elements in the runtime deployment descriptor.



application-client

The `application-client` element is the root element of a WebLogic-specific runtime 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.