



BEA WebLogic Server™

Developing Web Applications for WebLogic Server

Version 8.1
Revised: September 26, 2005

Copyright

Copyright © 2003-2005 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

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

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

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

Trademarks or Service Marks

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

All other trademarks are the property of their respective companies.

Contents

About This Document

Audience	xiv
e-docs Web Site	xiv
How to Print the Document	xiv
Related Information	xiv
Contact Us!	xv
Documentation Conventions	xv

1. Web Applications Basics

How to Use This Book	1-1
Overview of Web Applications	1-1
Servlets	1-2
Java Server Pages	1-3
Web Application Directory Structure	1-3
Main Steps to Create a Web Application	1-5
Step One: Create the Enterprise Application Wrapper	1-5
Step Two: Create the Web Application	1-6
Step Three: Creating the build.xml File	1-7
Step Four: Execute the Split Development Directory Structure Ant Tasks	1-7
URLs and Web Applications	1-8
Web Application Developer Tools	1-8
WebLogic Builder	1-8

Ant Tasks to Create Skeleton Deployment Descriptors	1-9
XML Editors	1-9
appc Compiler	1-9
appc Syntax	1-9
appc Options	1-10
wlappc Ant Task	1-10
Web Application Security	1-10

2. Deploying Web Applications

Deployment Overview	2-2
Deployment Descriptors	2-2
Using WebLogic Builder to Edit Deployment Descriptors	2-3
Manually Editing XML Deployment Files	2-3
DOCTYPE Header Information	2-4
Dynamic Descriptor Updates	2-4
Deployment Options	2-5
Deployment Guidelines for Web Applications	2-6
Deploying Web Applications as Part of an Enterprise Application	2-6
Auto-Deployment of Web Applications	2-6
Redeploying a Web Application in Exploded Directory Format	2-6
Using Command Line Deployment	2-7
Refreshing Static Components	2-8
Deploying Multiple Web Applications	2-8

3. Configuring Web Application Components

Configuring Servlets	3-1
Servlet Mapping	3-2
Servlet Initialization Attributes	3-5

Configuring Java Server Pages (JSPs)	3-5
Registering a JSP as a Servlet	3-6
Configuring JSP Tag Libraries	3-7
Configuring Welcome Pages	3-7
Setting Up a Default Servlet	3-8
Customizing HTTP Error Responses	3-9
Using CGI with WebLogic Server	3-9
Configuring WebLogic Server to Use CGI	3-9
Requesting a CGI Script	3-11
CGI Best Practices	3-11
Serving Resources from the CLASSPATH with the ClasspathServlet	3-12
Configuring Resources in a Web Application	3-12
Configuring External Resources	3-13
Referencing External EJBs	3-14
More about the ejb-ref* Elements	3-15
Referencing Application-Scoped EJBs	3-15
Loading Servlets, Context Listeners, and Filters	3-18
Determining the Encoding of an HTTP Request	3-19
Mapping IANA Character Sets to Java Character Sets	3-20

4. Using Sessions and Session Persistence in Web Applications

Overview of HTTP Sessions	4-1
Setting Up Session Management	4-1
HTTP Session Properties	4-2
Session Timeout	4-2
Configuring WebLogic Server Session Cookies	4-2
Configuring Application Cookies That Outlive a Session	4-3
Logging Out and Ending a Session	4-3

Configuring Session Persistence	4-4
Attributes Shared by Different Types of Session Persistence	4-4
Using Memory-based, Single-server, Non-replicated Persistent Storage	4-5
Using File-based Persistent Storage	4-5
Using a Database for Persistent Storage (JDBC persistence)	4-6
Using Cookie-Based Session Persistence	4-9
Using URL Rewriting Instead of Cookies	4-10
Coding Guidelines for URL Rewriting	4-10
URL Rewriting and Wireless Access Protocol (WAP)	4-11

5. Application Events and Event Listener Classes

Overview of Application Event Listener Classes	5-2
Servlet Context Events	5-2
HTTP Session Events	5-3
Configuring an Event Listener Class	5-4
Writing an Event Listener Class	5-5
Templates for Event Listener Classes	5-5
Servlet Context Event Listener Class Example	5-5
HTTP Session Attribute Event Listener Class Example	5-6
Additional Resources	5-7

6. Filters

Overview of Filters	6-1
How Filters Work	6-2
Uses for Filters	6-2
Writing a Filter Class	6-2
Configuring Filters	6-3
Configuring a Filter	6-3

Configuring a Chain of Filters	6-5
Filtering the Servlet Response Object	6-5
Additional Resources	6-6

A. web.xml Deployment Descriptor Elements

icon	A-2
display-name	A-2
description	A-3
distributable	A-3
context-param	A-4
filter	A-5
filter-mapping	A-7
listener	A-7
servlet	A-8
icon	A-9
init-param	A-9
security-role-ref	A-10
servlet-mapping	A-10
session-config	A-11
mime-mapping	A-12
welcome-file-list	A-12
error-page	A-13
taglib	A-13
resource-env-ref	A-14
resource-ref	A-15
security-constraint	A-16
web-resource-collection	A-17
auth-constraint	A-17

user-data-constraint	A-18
login-config	A-19
form-login-config	A-19
security-role	A-20
env-entry	A-20
ejb-ref	A-21
.	ejb-local-refA-22

B. weblogic.xml Deployment Descriptor Elements

description	B-2
weblogic-version	B-2
security-role-assignment	B-2
run-as-role-assignment	B-3
reference-descriptor	B-4
resource-env-description	B-4
resource-description	B-4
ejb-reference-description	B-5
session-descriptor	B-5
session-param	B-6
jsp-descriptor	B-11
JSP Attribute Names and Values	B-13
auth-filter	B-15
container-descriptor	B-15
check-auth-on-forward	B-15
redirect-with-absolute-url	B-16
index-directory-enabled	B-16
index-directory-sort-by	B-16
servlet-reload-check-secs	B-16

single-threaded-servlet-pool-size	B-16
session-monitoring-enabled	B-17
save-sessions-enabled	B-17
prefer-web-inf-classes	B-17
default-mime-type	B-17
retain-original-url	B-17
charset-params	B-17
input-charset	B-18
charset-mapping	B-18
virtual-directory-mapping	B-19
url-match-map	B-20
preprocessor	B-20
preprocessor-mapping	B-21
security-permission	B-21
context-root	B-22
wl-dispatch-policy	B-22
servlet-descriptor	B-22
init-as	B-23
destroy-as	B-24

About This Document

This document describes how to assemble and configure J2EE Web applications.

The document is organized as follows:

- [Chapter 1, “Web Applications Basics,”](#) is an overview of using Web applications in WebLogic Server.
- [Chapter 2, “Deploying Web Applications,”](#) discusses Web application deployment.
- [Chapter 3, “Configuring Web Application Components,”](#) describes how to configure Web application components.
- [Chapter 4, “Using Sessions and Session Persistence in Web Applications,”](#) describes how to use HTTP sessions and session persistence in a Web application.
- [Chapter 5, “Application Events and Event Listener Classes,”](#) describes how to use J2EE event listeners in a Web application.
- [Chapter 6, “Filters,”](#) describes how to use filters in a Web application.
- [Appendix A, “web.xml Deployment Descriptor Elements,”](#) provides a reference of deployment descriptor elements for the `web.xml` deployment descriptor.
- [Appendix B, “weblogic.xml Deployment Descriptor Elements,”](#) provides a reference of deployment descriptor elements for the `weblogic.xml` deployment descriptor.

Audience

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

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

- [Programming WebLogic HTTP Servlets](#)
- [Programming WebLogic Java Server Pages \(JSPs\)](#)
- [Programming WebLogic Web Services](#)
- [Developing WebLogic Server Applications](#)
- [Deploying WebLogic Server Applications](#)

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

Contact Us!

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
<i>italics</i>	Emphasis and book titles.

About This Document

Convention	Usage
monospace text	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard. <i>Examples:</i> <pre>import java.util.Enumeration; chmod u+w * samples/domains/examples/applications .java config.xml float</pre>
<i>monospace</i> <i>italic</i> text	Variables in code. <i>Example:</i> <pre>String CustomerName;</pre>
UPPERCASE TEXT	Device names, environment variables, and logical operators. <i>Examples:</i> <pre>LPT1 BEA_HOME OR</pre>
{ }	A set of choices in a syntax line.
[]	Optional items in a syntax line. <i>Example:</i> <pre>java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]</pre>
	Separates mutually exclusive choices in a syntax line. <i>Example:</i> <pre>java weblogic.deploy [list deploy undeploy update] password {application} {source}</pre>

Convention	Usage
...	Indicates one of the following in a command line: <ul style="list-style-type: none">• An argument can be repeated several times in the command line.• The statement omits additional optional arguments.• You can enter additional attributes, values, or other information
.	Indicates the omission of items from a code example or from a syntax line.
.	
.	

About This Document

Web Applications Basics

The following sections provide an overview of WebLogic Server Web applications:

- [“How to Use This Book”](#) on page 1-1
- [“Overview of Web Applications”](#) on page 1-1
- [“Main Steps to Create a Web Application”](#) on page 1-5
- [“URLs and Web Applications”](#) on page 1-8
- [“Web Application Developer Tools”](#) on page 1-8
- [“Web Application Security”](#) on page 1-10

How to Use This Book

As you develop and deploy your Web application, you will use this guide in conjunction with *Developing WebLogic Server Applications* and *Deploying WebLogic Server Applications*. These two guides provide detailed procedures and are your primary sources for creating, packaging, and deploying J2EE applications, including Web applications, to WebLogic Server. Refer to this guide, *Developing Web Applications for WebLogic Server*, to supplement that information with procedures and reference material that are specific to Web applications.

Overview of Web Applications

A Web application contains an application’s resources, such as servlets, JavaServer Pages (JSPs), JSP tag libraries, and any static resources such as HTML pages and image files. A Web

application can also define links to outside resources such as Enterprise JavaBeans (EJBs). Web applications deployed on WebLogic Server use a standard J2EE deployment descriptor file and a WebLogic-specific deployment descriptor file to define their resources and operating attributes.

JSPs and HTTP servlets can access all services and APIs available in WebLogic Server. These services include EJBs, database connections via Java Database Connectivity (JDBC), JavaMessaging Service (JMS), XML, and more.

A Web archive (WAR file) contains the files that make up a Web application (WAR file). A WAR file is deployed as a unit on one or more WebLogic Server instances.

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

- One servlet or Java Server Page (JSP), along with any helper classes.
- A `web.xml` deployment descriptor, which is a J2EE standard XML document that describes the contents of a WAR file.
- A `weblogic.xml` deployment descriptor, which is an XML document containing WebLogic Server-specific elements for Web applications.

A Web archive may also include HTML or XML pages and supporting files such as image and multimedia files.

The WAR file can be deployed alone or packaged in an Enterprise application archive (EAR file) with other application components. If deployed alone, the archive must end with a `.war` extension. If deployed in an EAR file, the archive must end with an `.ear` extension.

BEA recommends that you package and deploy your stand-alone Web applications as part of an Enterprise application. This is a BEA best practice, which allows for easier application migration, additions, and changes. Also, packaging your applications as part of an Enterprise application allows you to take advantage of the split development directory structure, which provides a number of benefits over the traditional single directory structure. See [“Creating WebLogic Server Applications”](#) in *Developing WebLogic Server Applications*.

Note: If you are deploying a directory in exploded format (not archived), do not name the directory `.ear`, `.jar`, and so on. For more information on archived format, see [“Web Application Directory Structure”](#) on page 1-3.

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.

Java Server Pages

Java Server Pages (JSPs) are Web pages coded with an extended HTML that makes it possible to embed Java code in a Web page. JSPs can call custom Java classes, called taglibs, using HTML-like tags. The WebLogic appc compiler `weblogic.appc` generates JSPs and validates descriptors.

You can also precompile JSPs and package the servlet class in the Web Archive to avoid compiling in the server. Servlets and JSPs may require additional helper classes to be deployed with the Web application.

Web Application Directory Structure

Web applications use a standard directory structure defined in the J2EE specification. You can deploy a Web application as a collection of files that use this directory structure, known as exploded directory format, or as an archived file called a WAR file. BEA recommends that you package and deploy your WAR file as part of an Enterprise application. This is a BEA best practice, which allows for easier application migration, additions, and changes. Also, packaging your Web application as part of an Enterprise application allows you to take advantage of the split development directory structure, which provides a number of benefits over the traditional single directory structure. See “[Creating WebLogic Server Applications](#)” in *Developing WebLogic Server Applications*.

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 `WEB-INF` directory contains the deployment descriptors for the Web application (`web.xml` and `weblogic.xml`) and two subdirectories for storing compiled Java classes and library JAR files. These subdirectories are respectively named `classes` and `lib`. JSP taglibs are stored in the `WEB-INF` directory at the top level of the staging directory. The Java classes include servlets, helper classes and, if desired, precompiled JSPs.

All servlets, classes, static files, and other resources belonging to a Web application are organized under a directory hierarchy.

DefaultWebApp/

Place your static files, such as HTML files and JSP files in the directory that is the document root of your Web application. In the default installation of WebLogic Server, this directory is called `DefaultWebApp`, under `user_domains/mydomain/applications`.

(To make your Web application the default Web application, you must set `context-root` to `"/"` in the `weblogic.xml` deployment descriptor file.)

DefaultWebApp/WEB-INF/web.xml

The Web application deployment descriptor that configures the Web application.

DefaultWebApp/WEB-INF/weblogic.xml

The WebLogic-specific deployment descriptor file that defines how named resources in the `web.xml` file are mapped to resources residing elsewhere in WebLogic Server. This file is also used to define JSP and HTTP session attributes.

DefaultWebApp/WEB-INF/classes

Contains server-side classes such as HTTP servlets and utility classes.

DefaultWebApp/WEB-INF/lib

Contains JAR files used by the Web application, including JSP tag libraries.

The entire directory, once staged, is bundled into a WAR file using the `jar` command. The WAR file can be deployed alone or as part of an Enterprise application (recommended) with other application components, including other Web applications, EJB components, and WebLogic Server components.

JSP pages and HTTP servlets can access all services and APIs available in WebLogic Server. These services include EJBs, database connections through Java Database Connectivity (JDBC), JavaMessaging Service (JMS), XML, and more.

The following is an example of a Web application directory structure, in which `myWebApp/` is the staging directory:

Listing 1-1 Web Application Directory Structure

```
myWebApp/
    WEB-INF/
        web.xml
        weblogic.xml
    lib/
        MyLib.jar
    classes/
        MyPackage/
            MyServlet.class
    index.html
    index.jsp
```

Main Steps to Create a Web Application

The following steps summarize the procedure for creating a Web application as part of an Enterprise application using the split development directory structure. See “[Creating WebLogic Server Applications](#)” in *Developing WebLogic Server Applications*.

You may want to use developer tools included with WebLogic Server for creating and configuring Web applications. See “[Web Application Developer Tools](#)” on page 1-8.

Step One: Create the Enterprise Application Wrapper

1. Create a directory for your root EAR file:

```
\src\myEAR\
```

2. Set your environment as follows:

- On Windows NT, execute the `setWLSEnv.cmd` command, located in the directory `server\bin\`, where `server` is the top-level directory in which WebLogic Server is installed.
 - On UNIX, execute the `setWLSEnv.sh` command, located in the directory `server/bin/`, where `server` is the top-level directory in which WebLogic Server is installed and `domain` refers to the name of your domain.
3. Package your Enterprise application in the `\src\myEAR\` directory as follows:
- a. Place the Enterprise applications descriptors (`application.xml` and `weblogic-application.xml`) in the `META-INF\` directory. See [“Enterprise Application Deployment Descriptors”](#) in *Developing WebLogic Server Applications*.

You can automatically generate the Enterprise application descriptors using the DDInit Java utility by executing the following command:

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


For more information on DDInit, see [“Using the WebLogic Server Java Utilities.”](#)
 - b. Edit the deployment descriptors as needed to fine-tune the behavior of your Enterprise application. See [“Deployment Descriptors”](#) on page 2-2.
 - c. Place the Enterprise application `.jar` files in:

```
\src\myEAR\APP-INF\lib\
```

Step Two: Create the Web Application

1. Create a directory for your Web application in the the root of your EAR file:

```
\src\myEAR\myWebApp
```
2. Package your Web application in the `\src\myEAR\myWebApp\` directory as follows:
- a. Place the Web application descriptors (`web.xml` and `weblogic.xml`) in the `\src\myEAR\myWebApp\WEB-INF\` directory. See [Appendix A, “web.xml Deployment Descriptor Elements,”](#) and [Appendix B, “weblogic.xml Deployment Descriptor Elements.”](#)

You can automatically generate the Web application descriptors using the DDInit utility by executing the following command:

```
java weblogic.marathon.ddinit.WebInit \myEAR\myWebApp
```


For more information on DDInit, see [“Using the WebLogic Server Java Utilities.”](#)

- b. Edit the deployment descriptors as needed to fine-tune the behavior of your Enterprise application. See [“Deployment Descriptors” on page 2-2](#).
- c. Place all HTML files, JSPs, images and any other files referenced by the Web application pages in the root of the Web application:

```
\src\myEAR\myWebApp\images\myimage.jpg
```

```
\src\myEAR\myWebApp\login.jsp
```

```
\src\myEAR\myWebApp\index.html
```

- d. Place your Web application Java source files (servlets, tag libs, other classes referenced by servlets or tag libs) in:

```
\src\myEAR\myWebApp\WEB-INF\src\
```

Step Three: Creating the build.xml File

Once you have set up your directory structure, you create the `build.xml` file using the `weblogic.BuildXMLGen` utility. See [“Creating WebLogic Server Applications”](#) in *Developing WebLogic Server Applications*.

Step Four: Execute the Split Development Directory Structure Ant Tasks

1. Execute the `wlcompile` Ant task to invoke the `javac` compiler. This compiles your Web application Java components into an output directory: `/build/myEAR/WEB-INF/classes`. See [“Creating WebLogic Server Applications”](#) in *Developing WebLogic Server Applications*.
2. Execute `wlappc` Ant task to invoke the `appc` compiler. This compiles any JSPs and container-specific EJB classes for deployment. See [“Creating WebLogic Server Applications”](#) in *Developing WebLogic Server Applications*. Also see [“appc Compiler” on page 1-9](#).
3. Execute the `wldeploy` Ant task to deploy your Web application as part of an archived or exploded EAR to WebLogic Server. See [“Deployment Tools Reference”](#) in *Deploying WebLogic Server Applications*.
4. If this is a production environment (rather than development), execute the `wlpackage` Ant task to package your Web application as part of an archived or exploded EAR. See [“Creating WebLogic Server Applications”](#) in *Developing WebLogic Server Applications*.

Note: The `wlpackage` Ant task places compiled versions of your Java source files in the `build` directory. For example: `/build/myEAR/myWebApp/classes`.

URLs and Web Applications

You construct the URL that a client uses to access a Web application using the following pattern:

`http://hoststring/ContextPath/servletPath/pathInfo`

Where

hoststring

is either a host name that is mapped to a virtual host or `hostname:portNumber`.

ContextPath

is the name of your Web application.

servletPath

is a servlet that is mapped to the `servletPath`.

pathInfo

is the remaining portion of the URL, typically a file name.

If you are using virtual hosting, you can substitute the virtual host name for the *hoststring* portion of the URL.

Web Application Developer Tools

BEA provides several tools to help you create and configure Web applications.

WebLogic Builder

WebLogic Builder is a graphical tool for assembling a J2EE application module; creating and editing its deployment descriptors; and deploying it to WebLogic Server.

WebLogic Builder is a graphical environment in which you edit an application's deployment descriptor XML files. You can view these XML files as you edit them graphically in WebLogic Builder, but you won't need to make textual edits to the XML files.

Use WebLogic Builder to do the following development tasks:

- Generate deployment descriptor files for a J2EE module
- Edit a module's deployment descriptor files
- Compile and validate deployment descriptor files
- Deploy a J2EE application to a server

For more information on WebLogic Builder, see [WebLogic Builder Online Help](#).

Ant Tasks to Create Skeleton Deployment Descriptors

You can use the WebLogic Ant utilities to create skeleton deployment descriptors. These utilities are Java classes shipped with your WebLogic Server distribution. The Ant task looks at a directory containing a Web application and creates deployment descriptors based on the files it finds in the Web application. Because the Ant utility does not have information about all desired configurations and mappings for your Web application, the skeleton deployment descriptors the utility creates are incomplete. After the utility creates the skeleton deployment descriptors, you can use a text editor, an XML editor, or the Administration Console to edit the deployment descriptors and complete the configuration of your Web application.

For more information on using Ant utilities to create deployment descriptors, see "[Tools for Deploying](#)" in *Deploying WebLogic Server Applications*.

XML Editors

To create and edit XML files, you can use an XML Editor with DTD validation, such as BEA XML Editor on dev2dev or XMLSpy. (An evaluation copy of XMLSpy is bundled with this version of WebLogic Server.) See *BEA dev2dev Online* at <http://dev2dev.bea.com/index.jsp>.

appc Compiler

The appc compiler compiles and generates J2EE EAR files, EJB JAR files, and Web application WAR files for deployment. It also validates the descriptors for compliance with the current specifications at both the individual module level and the application level. The application level checks include checks between the application-level deployment descriptors and the individual modules as well as validation checks across the modules. The appc compiler reports any warnings or errors encountered in the descriptors. Finally, the appc compiler compiles all of the relevant modules into an EAR file, which can be deployed to WebLogic Server.

appc Syntax

Use the following syntax to run appc:

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

For example, to validate the descriptors in `myWebApp.war`, as well as compile the `jsp` files and update the WAR file with the resulting class files, you would execute the following command:

```
java weblogic.appc -keepgenerated myWebApp.war
```

appc Options

For a complete list of `appc` options, see ["Compiling Java Code"](#) in *Developing WebLogic Server Applications*.

wlappc Ant Task

Use the `wlappc` Ant task to invoke the `appc` compiler:

```
<wlappc source="${dest.dir}" />
```

For a complete list of `wlappc` options, see ["Compiling Java Code"](#) in *Developing WebLogic Server Applications*.

Web Application Security

You can secure a Web application by restricting access to certain URL patterns in the Web application or programmatically using security calls in your servlet code.

At runtime, your username and password are authenticated using the applicable security realm for the Web application. Authorization is verified according to the security constraints configured in `web.xml` or the external policies that might have been created using Administration Console for the Web application. For information on creating policies using the Administration Console, see the Online Help.

At runtime, the WebLogic Server active security realm applies the Web application security constraints to the specified Web application resources. Note that a security realm is shared across multiple virtual hosts.

For detailed instructions and an example on configuring security in Web applications, see [Securing WebLogic Resources](#). For more information on WebLogic security, refer to [Programming WebLogic Security](#).

Deploying Web Applications

This chapter discusses the deployment of Web applications:

- “Deployment Overview” on page 2-2
- “Deployment Descriptors” on page 2-2
- “Deployment Options” on page 2-5
- “Deployment Guidelines for Web Applications” on page 2-6

WebLogic Server application deployment is covered in more detail in *Deploying WebLogic Server Applications*. This topics covered in this section explain only deployment procedures that are specific to Web applications.

Deployment Overview

Deployment of a Web application is similar to deployment of Connectors, EJBs, and Enterprise Applications. Like these deployment units, you can deploy a Web application in an exploded directory format or as an archive file.

Note: For development and production purposes, BEA recommends that you deploy applications in exploded (unarchived) directory format. This applies also to stand-alone Web applications, EJBs, and connectors packaged as part of an Enterprise application. Using this format allows you to update files directly in the exploded directory rather than having to unarchive, edit, and rearchive the whole application. Using exploded archive directories also has other benefits, as described in [Exploded Archive Directories](#) in *Deploying WebLogic Server Applications*.

Deploying a Web application enables WebLogic Server to serve the components of a Web application to clients. You can deploy a Web application using one of several procedures, depending on your environment and whether or not your Web application is in production. You can use the WebLogic Server Administration Console, the `weblogic.Deployer` utility, or you can use auto-deployment.

In the procedures for deploying a Web application, it is assumed that you have created a functional Web application that uses the correct directory structure and contains the `web.xml` deployment descriptor and, if needed, the `weblogic.xml` deployment descriptor. For an overview of the steps required to package and create a Web application, see [“Main Steps to Create a Web Application”](#) on page 1-5.

Deployment Descriptors

Web applications use two deployment descriptors to define their operational attributes. A `web.xml` deployment descriptor is a J2EE standard XML document that describes the contents of a WAR file. The `weblogic.xml` deployment descriptor is an XML document containing WebLogic Server-specific elements for Web applications.

For more information about the elements in these deployment descriptor, refer to [Appendix A](#), “`web.xml` Deployment Descriptor Elements,” and [Appendix B](#), “`weblogic.xml` Deployment Descriptor Elements.”

You can modify deployment descriptors using the following tools:

- **WebLogic Builder.** WebLogic Builder is the BEA preferred WebLogic Server tool for generating and editing deployment descriptors for WebLogic Server applications. It can also deploy WebLogic Server applications to single servers. For more information, see

“Deployment Tools Reference” in *Deploying WebLogic Server Applications*. For more information, see “Using WebLogic Builder to Edit Deployment Descriptors” on page 2-3.

- XML Editor with DTD validation, such as BEA XML Editor on dev2dev or XMLSpy. (An evaluation copy of XMLSpy is bundled with this version of WebLogic Server.) See [BEA dev2dev Online](http://dev2dev.bea.com/index.jsp) at <http://dev2dev.bea.com/index.jsp>.
- Using the Administration Console Descriptor tab. The Descriptor tab has replaced the deprecated Deployment Descriptor Editor in the Administration Console. For more information on the Descriptor tab, see the WebLogic Server online help. Also refer to “Dynamic Descriptor Updates” on page 2-4.

Using WebLogic Builder to Edit Deployment Descriptors

WebLogic Builder provides a visual environment for editing an application’s deployment descriptor XML files. You can view these XML files as you visually edit them in WebLogic Builder, but you do not need to make textual edits to the XML files.

Use WebLogic Builder for the following development tasks:

- Generate deployment descriptor files for a J2EE module
- Edit a module’s deployment descriptor files
- View deployment descriptor files
- Compile and validate deployment descriptor files
- Deploy a module to a server

For instructions on using WebLogic Builder, refer to the [WebLogic Builder](#) documentation.

Manually Editing XML Deployment Files

To manually edit XML elements:

- Make sure that you use an ASCII text editor that does not reformat the XML or insert additional characters that could invalidate the file.
- Use the correct case for file and directory names, even if your operating system ignores the case.
- To use the default value for an optional element, you can either omit the entire element definition or specify a blank value. For example:

```
<max-config-property></max-config-property>
```

DOCTYPE Header Information

When editing or creating XML deployment files, it is critical to include the correct `DOCTYPE` header for each deployment file. In particular, using an incorrect `PUBLIC` element within the `DOCTYPE` header can result in parser errors that may be difficult to diagnose.

The header refers to the location and version of the Document Type Definition (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 `weblogic.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.

XML files with incorrect header information may yield error messages similar to the following, when used with a utility that parses the XML (such as `appc`):

```
SAXException: This document may not have the identifier 'identifier_name'
identifier_name generally includes the invalid text from the PUBLIC element.
```

The `DOCTYPE` header for the `weblogic.xml` file is as follows:

```
<!DOCTYPE weblogic-web-app PUBLIC
    "-//BEA Systems, Inc.//DTD Web Application 8.1//EN"
    "http://www.bea.com/servers/wls810/dtd/weblogic810-web-jar.dtd">
```

Dynamic Descriptor Updates

This release of WebLogic Server has deprecated the Administration Console Deployment Descriptor Editor. A new Descriptor tab in the Administration Console has replaced it. The Descriptor tab is intended for administrative purposes.

Using the Descriptor tab, you can view, modify, and persist deployment descriptor elements to the descriptor file within the Web application. You can also the apply descriptor file changes to the deployed application on all its targets.

However, these changes take effect dynamically at runtime without requiring the Web application to be redeployed. The descriptor element attributes contained in the Descriptor tab are limited to only those that may be dynamically changed at runtime. These include the following:

- **Session Cookie Max Age Secs**—The life span of the session cookie, in seconds, after which it expires on the client.

- **Session Invalidation Interval Secs**—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.
- **Session Timeout Secs**—The amount of time (in seconds) that a session can remain inactive before it is invalidated.
- **Servlet Reload Check Secs**—The amount of time (in seconds) that WebLogic Server waits to check if a servlet was modified and needs to be reloaded.
- **Single Threaded Servlet Pool Size**—The size of the pool used for single-threaded mode servlets.
- **Index Directory Enabled**—Specifies whether the target should automatically generate an HTML directory listing if no suitable index file is found.
- **Session Monitoring Enabled**—Specifies whether runtime MBeans will be created for session monitoring.
- **JSPPage Check Secs**—The interval, in seconds, at which WebLogic Server checks to see if JSP files have changed and need recompiling.
- **JSPKeep Generated**—Specifies whether to save the Java files that are generated as an intermediary step in the JSP compilation process.
- **JSPVerbose**—Specifies whether to print debugging info to the browser during compilation.
- **Enable JSP Line Numbers**—Specifies whether to add JSP line numbers to generated class files to aid in debugging.

Deployment Options

You can deploy a Web application by using:

- WebLogic Server Administration Console
- `weblogic.Deployer` Utility
- Auto-deployment. This is useful for testing purposes. For Web-application specific information on this topic, see [“Auto-Deployment of Web Applications”](#) on page 2-6

For more information on these tools, see [“Deployment Tools Reference”](#) in *Deploying WebLogic Server Applications*.

Deployment Guidelines for Web Applications

The following sections provide some guidelines that are specific to deploying Web applications.

Deploying Web Applications as Part of an Enterprise Application

For both production and development purposes, BEA recommends that you package and deploy your stand-alone Web applications, EJBs, and resource adapters as part of an Enterprise application. This practice allows for easier application migration, additions, and changes. For example, to add more than one Web application, EJB, or resource adapter module to an application, you must package the modules in an Enterprise application. Packaging and deploying stand-alone applications in an Enterprise application from the start saves you a lot of time. It also allows you to take advantage of the split development directory structure, which provides a number of benefits over the traditional single directory structure.

For more information, see “[Creating WebLogic Server Applications](#)” in *Developing WebLogic Server Applications*.

Auto-Deployment of Web Applications

When WebLogic Server is running in development mode, Web applications that are copied into the `applications/` directory are automatically deployed. In addition, modifications to files within these applications can trigger either partial or full redeployment.

Auto-deployment of the `applications/` directory is intended for iterative development (not for production) and only for single server (specifically, the Administration Server). That is, applications deployed using the `applications/` directory should not be targeted to servers other than the Administration Server.

Redeploying a Web Application in Exploded Directory Format

When a Web application is deployed in exploded directory format, redeployment is affected by changing specific files within the directory structure. If the Web application is deployed using auto-deployment, you must modify a special file named `REDEPLOY`, which is located in the `WEB-INF/` directory of your Web application, to redeploy the entire application.

Modifying individual files of an exploded Web application can cause partial redeployment of the Web application, depending on the files you changed. For example, modifying servlets or other classes in the `WEB-INF/classes` directory can cause the associated classloader to be replaced,

so that the application uses the latest version of the class. (This is one reason why objects stored in a session should be serializable.) You can control the timing of automatic class updates by setting the `servlet-reload-check-secs` element, as described in [Appendix B, “weblogic.xml Deployment Descriptor Elements.”](#)

To re-deploy an entire auto-deployed Web application in exploded directory format, modify the `REDEPLOY` file:

1. Create an empty file called `REDEPLOY` and place it in the `WEB-INF` directory of your Web application. (You may have to create this directory.)
2. Open the `REDEPLOY` file, modify the contents (this modification can be as simple as adding a space character in the file contents—any modification works), and then save the file.

Note: The point in modifying the contents of `REDEPLOY` is to change the timestamp of the file. You can achieve this by making a simple edit and saving it.

3. Alternatively, on UNIX machines, use the `touch` command on the `REDEPLOY` file. For example:

```
touch user_domains/mydomain/applications/DefaultWebApp/WEB-INF/REDEPLOY
```

As soon as the `REDEPLOY` file is modified, the Web application is redeployed.

The servlet container detects changes to servlet and filter classes and resets its classloader when a change is detected. Note that all servlets and filters are reloaded together, so changing one servlet or filter causes all to be reloaded. For more information, refer to [“Using Command Line Deployment” on page 2-7.](#)

Similarly, changes to JSP files cause a recompilation of the JSP. JSP class files are reloaded individually, so reloading a JSP only affects the one JSP class.

Because static content is served directly from the disk, no redeployment is necessary for HTML pages, image files, and so on. However, if you are using caching tags or filters, you must ensure that the cache is flushed when modifying content.

You can redeploy a Web application deployed in exploded directory format when using auto-deployment by modifying a special file called `REDEPLOY`. You can also cause a partial redeploy by copying a new version of a class file over an old in the `WEB-INF/classes` directory.

Using Command Line Deployment

You use command line deployment for applications that are not auto-deployed. This includes any applications that are deployed from locations other than the `applications/` directory. In

addition, auto-deployment is disabled when the server is running in production mode. Therefore, command line deployment is necessary for all applications when production mode is enabled.

You enable production mode by starting the domain's Administration Server using the following flag: `-Dweblogic.ProductionModeEnabled=true`. This sets the production mode for all server instances in the domain.

When you modify a component (for instance, a servlet, JSP, or HTML page) of a Web application, you must take additional steps to refresh the modified component so that it is also deployed on any targeted Managed Servers. One way to refresh a component is to redeploy the entire Web application. Redeploying the Web application means that the entire Web application (not just the modified component) is re-sent over the network to all of the Managed Servers targeted by that Web application.

Note the following regarding re-deployment of Web applications:

- Depending on your environment, there may be performance implications due to increased network traffic when a Web application is re-sent to the Managed Servers.
- If the Web application is currently in production and in use, redeploying the Web application causes WebLogic Server to lose all active HTTP sessions for current users of the Web application.
Note: Sessions will be restored if you have set the `save-sessions-enabled` element to `true` in the `weblogic.xml` deployment descriptors.
- If you have updated any Java class files, you must redeploy the entire Web application to refresh the class.
- If you change the deployment descriptors, you must redeploy the entire Web application.

Refreshing Static Components

BEA recommends that you use the `weblogic.Deployer` utility to refresh static components (such as JSP files, HTML files, image files, and so on). For information about the `weblogic.Deployer` utility, see “[Deployment Tools Reference](#)” in *Deploying WebLogic Server Applications*.

Deploying Multiple Web Applications

In order to deploy multiple Web applications, context roots must be unique. For example, `/test-1`, `/test-2`, and so on. This prevents the context root display for Web application modules from becoming corrupted as the number of Web application modules grows.

Configuring Web Application Components

The following sections describe how to configure Web application components:

- “Configuring Servlets” on page 3-1
- “Configuring Java Server Pages (JSPs)” on page 3-5
- “Configuring JSP Tag Libraries” on page 3-7
- “Configuring Welcome Pages” on page 3-7
- “Setting Up a Default Servlet” on page 3-8
- “Customizing HTTP Error Responses” on page 3-9
- “Using CGI with WebLogic Server” on page 3-9
- “Serving Resources from the CLASSPATH with the ClasspathServlet” on page 3-12
- “Configuring Resources in a Web Application” on page 3-12
- “Loading Servlets, Context Listeners, and Filters” on page 3-18
- “Mapping IANA Character Sets to Java Character Sets” on page 3-20

Configuring Servlets

You define servlets as a part of a Web application in several entries in the Web Application deployment descriptor. The first entry, under the `<servlet>` element, defines a name for the servlet and specifies the compiled class that executes the servlet. (Or, instead of specifying a

Servlet class, you can specify a JSP.) This element also contains definitions for initialization attributes and security roles for the servlet. The second entry, under the `<servlet-mapping>` element, defines the URL pattern that calls this servlet.

Servlet Mapping

Servlet mapping controls how you access a servlet. The following examples demonstrate how you can use servlet mapping in your Web application. In the examples, a set of servlet configurations and mappings (from the `web.xml` deployment descriptor) is followed by a table (see “[url-patterns and Servlet Invocation](#)” on page 3-3) showing the URLs used to invoke these servlets.

For more information on servlet mappings, such as general servlet mapping rules and conventions, refer to Section 11 of the Servlet 2.3 specification at:

<http://www.jsp.org/aboutJava/communityprocess/final/jsr053/>

Listing 3-1 Servlet Mapping Example

```
<servlet>
  <servlet-name>watermelon</servlet-name>
  <servlet-class>myservlets.watermelon</servlet-class>
</servlet>

<servlet>
  <servlet-name>garden</servlet-name>
  <servlet-class>myservlets.garden</servlet-class>
</servlet>

<servlet>
  <servlet-name>list</servlet-name>
  <servlet-class>myservlets.list</servlet-class>
</servlet>

<servlet>
  <servlet-name>kiwi</servlet-name>
  <servlet-class>myservlets.kiwi</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>watermelon</servlet-name>
```

```

    <url-pattern>/fruit/summer/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>garden</servlet-name>
    <url-pattern>/seeds/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>list</servlet-name>
    <url-pattern>/seedlist</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>kiwi</servlet-name>
    <url-pattern>*.abc</url-pattern>
</servlet-mapping>

```

Table 3-1 url-patterns and Servlet Invocation

URL	Servlet Invoked
http://host:port/mywebapp/fruit/summer/index.html	watermelon
http://host:port/mywebapp/fruit/summer/index.abc	watermelon
http://host:port/mywebapp/seedlist	list

Table 3-1 url-patterns and Servlet Invocation

URL	Servlet Invoked
http://host:port/mywebapp/seedlist/index.html	<p>The default servlet, if configured, or an HTTP 404 File Not Found error message.</p> <p>If the mapping for the list servlet had been /seedlist*, the list servlet would be invoked.</p>
http://host:port/mywebapp/seedlist/pear.abc	<p>kiwi</p> <p>If the mapping for the list servlet had been /seedlist*, the list servlet would be invoked.</p>
http://host:port/mywebapp/seeds	garden
http://host:port/mywebapp/seeds/index.html	garden
http://host:port/mywebapp/index.abc	kiwi

ServletServlet can be used to create a default mappings for servlets. For example, to create a default mapping to map all servlets to /myservlet/*, so the servlets can be called using http://host:port/web-app-name/myservlet/com/foo/Servlet, add the following to your web.xml file:

```
<servlet>
  <servlet-name>ServletServlet</servlet-name>
  <servlet-class>weblogic.servlet.ServletServlet</servlet-class>
</servlet>
```

```

<servlet-mapping>
<servlet-name>ServletServlet</servlet-name>
  <url-pattern>myservlet/*</url-pattern>
</servlet-mapping>

```

Servlet Initialization Attributes

You define initialization attributes for servlets in the Web Application deployment descriptor, `web.xml`, in the `<init-param>` element of the `<servlet>` element, using `<param-name>` and `<param-value>` tags. For example:

Listing 3-2 Example of Configuring Servlet Initialization Attributes in `web.xml`

```

<servlet>
  <servlet-name>HelloWorld2</servlet-name>
  <servlet-class>examples.servlets.HelloWorld2</servlet-class>

  <init-param>
    <param-name>greeting</param-name>
    <param-value>Welcome</param-value>
  </init-param>

  <init-param>
    <param-name>person</param-name>
    <param-value>WebLogic Developer</param-value>
  </init-param>
</servlet>

```

For more information on editing the Web Application deployment descriptor, see [“Deploying Web Applications” on page 2-1](#).

Configuring Java Server Pages (JSPs)

In order to deploy Java Server Pages (JSP) files, you must place them in the root (or in a subdirectory below the root) of a Web application. You define JSP configuration attributes in the

`<jsp-descriptor>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. These attributes define the following functionality:

- Options for the JSP compiler
- Debugging
- How often WebLogic Server checks for updated JSPs that need to be recompiled
- Character encoding

For a complete description of these attributes, see [“JSP Attribute Names and Values” on page B-13](#).

Registering a JSP as a Servlet

You can register a JSP as a servlet using the `<servlet>` element. A servlet container maintains a map of the servlets known to it. This map is used to resolve requests that are made to the container. Adding entries into this map is known as "registering" a servlet. You add entries to this map by referencing a `<servlet>` element in `web.xml` through the `<servlet-mapping>` entry.

A JSP is a type of servlet; registering a JSP is a special case of registering a servlet. Normally, JSPs are implicitly registered the first time you invoke them, based on the name of the JSP file. Therefore, the `myJSPfile.jsp` file would be registered as `myJSPfile.jsp` in the mapping table. You can implicitly register JSPs, as illustrated in the following example. In this example, you request the JSP with the name `/main` instead of the implicit name `myJSPfile.jsp`.

In this example, a URL containing `/main` will invoke `myJSPfile.jsp`:

```
<servlet>
    <servlet-name>myFoo</servlet-name>
    <jsp-file>myJSPfile.jsp</jsp-file>
</servlet>
<servlet-mapping>
    <servlet-name>myFoo</servlet-name>
    <url-pattern>/main</url-pattern>
</servlet-mapping>
```

Registering a JSP as a servlet allows you to specify the load order, initialization attributes, and security roles for a JSP, just as you would for a servlet.

Configuring JSP Tag Libraries

Weblogic Server lets you create and use custom JSP tags. Custom JSP tags are Java classes you can call from within a JSP page. To create custom JSP tags, you place them in a tag library and define their behavior in a tag library descriptor (TLD) file. You make this TLD available to the Web application containing the JSP by defining it in the Web Application deployment descriptor. It is a good idea to place the TLD file in the `WEB-INF` directory of your Web application, because that directory is never available publicly.

In the Web Application deployment descriptor, you define a URI pattern for the tag library. This URI pattern must match the value in the `taglib` directive in your JSP pages. You also define the location of the TLD. For example, if the `taglib` directive in the JSP page is:

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

and the TLD is located in the `WEB-INF` directory of your Web application, you would create the following entry in the Web Application deployment descriptor:

```
<taglib>
  <taglib-uri>myTaglib</taglib-uri>
  <taglib-location>WEB-INF/myTLD.tld</taglib-location>
</taglib>
```

You can also deploy a tag library as a `.jar` file.

For more information on creating custom JSP tag libraries, see [Programming JSP Tag Extensions](#).

WebLogic Server also includes several custom JSP tags that you can use in your applications. These tags perform caching, facilitate query attribute-based flow control, and facilitate iterations over sets of objects. For more information, see:

- “Using Custom WebLogic JSP Tags” in [Programming WebLogic JSP](#).
- “Using WebLogic JSP Form Validation Tags” in [Programming WebLogic JSP](#).

Configuring Welcome Pages

WebLogic Server allows you to set a page that is served by default if the requested URL is a directory. This feature can make your site easier to use, because the user can type a URL without giving a specific filename.

Welcome pages are defined at the Web application level. If your server is hosting multiple Web applications, you need to define welcome pages separately for each Web application.

To define Welcome pages, you edit the Web application deployment descriptor, `web.xml`. If you do not define Welcome Pages, WebLogic Server looks for the following files in the following order and serves the first one it finds:

1. `index.html`
2. `index.htm`
3. `index.jsp`

The following is an example Welcome page configuration:

Listing 3-3 Welcome Page Example

```
<servlet>
    <servlet-name>WelcomeServlet</servlet-name>
    <servlet-class>foo.bar.WelcomeServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>WelcomeServlet</servlet-name>
    <url-pattern>*.foo</url-pattern>
</servlet-mapping>

<welcome-file-list>
    <welcome-file>/welcome.foo</welcome-file>
</welcome-file-list>
```

Setting Up a Default Servlet

Each Web application has a *default servlet*. This default servlet can be a servlet that you specify, or, if you do not specify a default servlet, WebLogic Server uses an internal servlet called the `FileServlet` as the default servlet.

You can register any servlet as the default servlet. Writing your own default servlet allows you to use your own logic to decide how to handle a request that falls back to the default servlet.

Setting up a default servlet replaces the `FileServlet` and should be done carefully because the `FileServlet` is used to serve most files, such as text files, HTML file, image files, and more. If

you expect your default servlet to serve such files, you will need to write that functionality into your default servlet.

To set up a user-defined default servlet:

1. Define your servlet as described in [“Configuring Servlets” on page 3-1](#).
2. If you still want the `FileServlet` to serve files with other extensions:
 - a. Define a servlet and give it a `<servlet-name>`, for example `myFileServlet`.
 - b. Define the `<servlet-class>` as `weblogic.servlet.FileServlet`.
 - a. Using the `<servlet-mapping>` element, map file extensions to the `myFileServlet` (in addition to the mappings for your default servlet). For example, if you want the `myFileServlet` to serve `.gif` files, map `*.gif` to the `myFileServlet`.

Note: The `FileServlet` includes the `SERVLET_PATH` when determining the source filename if `docHome` is not specified. As a result, it is possible to explicitly serve only files from specific directories by mapping the `FileServlet` to `/dir/*`, etc.

Customizing HTTP Error Responses

You can configure WebLogic Server to respond with your own custom Web pages or other HTTP resources when particular HTTP errors or Java exceptions occur, instead of responding with the standard WebLogic Server error response pages.

You define custom error pages in the `<error-page>` element of the Web application deployment descriptor (`web.xml`). For more information on error pages, see [“error-page” on page A-13](#).

Using CGI with WebLogic Server

Note: WebLogic Server provides functionality to support your legacy Common Gateway Interface (CGI) scripts. For new projects, we suggest you use HTTP servlets or JavaServer Pages.

WebLogic Server supports all CGI scripts through an internal WebLogic servlet called the `CGIServlet`. To use CGI, register the `CGIServlet` in the Web application deployment descriptor. For more information, see [“Configuring Servlets” on page 3-1](#).

Configuring WebLogic Server to Use CGI

To configure CGI in WebLogic Server:

1. Declare the `CGIServlet` in your Web application by using the `<servlet>` and `<servlet-mapping>` elements. The class name for the `CGIServlet` is `weblogic.servlet.CGIServlet`. You do not need to package this class in your Web application.
2. Register the following initialization attributes for the `CGIServlet` by defining the following `<init-param>` elements:

`cgiDir`

The path to the directory containing your CGI scripts. You can specify multiple directories, separated by a semicolon (;). This separator applies to both Windows and UNIX. If you do not specify `cgiDir`, the directory defaults to a directory named `cgi-bin` under the Web application root.

`useByteStream`

The alternate to using the default `Char` streams for data transfer, this parameter, which is case sensitive, will allow the use of images in the CGI servlet without distortion.

extension mapping

Maps a file extension to the interpreter or executable that runs the script. If the script does not require an executable, this initialization attribute may be omitted.

The `<param-name>` for extension mappings must begin with an asterisk followed by a dot, followed by the file extension, for example, `*.pl`.

The `<param-value>` contains the path to the interpreter or executable that runs the script. You can create multiple mappings by creating a separate `<init-param>` element for each mapping.

Listing 3-4 Example Web Application Deployment Descriptor Entries for Registering the `CGIServlet`

```
<servlet>
<servlet-name>CGIServlet</servlet-name>
<servlet-class>weblogic.servlet.CGIServlet</servlet-class>
<init-param>
  <param-name>cgiDir</param-name>
  <param-value>
    /bea/wlserver6.0/config/mydomain/applications/myWebApp/cgi-bin
  </param-value>
</init-param>
```

```

<init-param>
  <param-name>*.pl</param-name>
  <param-value>/bin/perl.exe</param-value>
</init-param>
</servlet>

...

<servlet-mapping>
  <servlet-name>CGIServlet</servlet-name>
  <url-pattern>/cgi-bin/*</url-pattern>
</servlet-mapping>

```

Requesting a CGI Script

The URL used to request a Perl script must follow the pattern:

```
http://host:port/myWebApp/cgi-bin/myscript.pl
```

Where

host:port

Is the host name and port number of WebLogic Server.

myWebApp

is the name of your Web application.

cgi-bin

is the url-pattern name mapped to the CGIServlet.

myscript.pl

is the name of the Perl script that is located in the directory specified by the `cgiDir` initialization attribute.

CGI Best Practices

The following are CGI best practices with respect to calling a subscript:

- You can use `sh subscript.sh` for both exploded (unarchived) Web applications and archived Web applications (WAR files).
- You can use `sh $PWD/subscript.sh` for both exploded (unarchived) Web applications and archived Web applications (WAR files).

Serving Resources from the CLASSPATH with the ClasspathServlet

If you need to serve classes or other resources from the system CLASSPATH, or from the WEB-INF/classes directory of a Web application, you can use a special servlet called the ClasspathServlet. The ClasspathServlet is useful for applications that use applets or RMI clients and require access to server-side classes. The ClasspathServlet is implicitly registered and available from any application.

The ClasspathServlet is always enabled by default. To disable it, set the ServerMBean parameter ClassPathServletDisabled to true (default = false).

The ClasspathServlet returns the classes or resources from the system CLASSPATH in the following order:

1. WEB-INF/classes
2. jar files under WEB-INF/lib/*
3. system CLASSPATH

To serve a resource from the WEB-INF/classes directory of a Web application, call the resource with a URL such as:

```
http://server:port/myWebApp/classes/my/resource/myClass.class
```

In this case, the resource is located in the following directory, relative to the root of the Web application:

```
WEB-INF/classes/my/resource/myClass.class
```

Warning: Because the ClasspathServlet serves any resource located in the system CLASSPATH, do not place resources that should not be publicly available in the system CLASSPATH.

Configuring Resources in a Web Application

The resources that you use in a Web application are generally deployed externally to the application. JDBC DataSources can optionally be deployed within the scope of the Web application as part of an EAR file.

Prior to WebLogic Server 7.0, JDBC DataSources were always deployed externally to the Web application. To use external resources in the Web application, you resolve the JNDI resource name that the application uses with the global JNDI resource name using the web.xml and

`weblogic.xml` deployment descriptors. See [“Configuring External Resources” on page 3-13](#) for more information.

WebLogic Server versions 7.x and later enable you to deploy JDBC DataSources as part of the Web application EAR file by configuring those resources in the `weblogic-application.xml` deployment descriptor. Resources deployed as part of the EAR file are referred to as *application-scoped* resources. These resources remain private to the Web application, and application components can access the resource names directly from the local JNDI tree at `java:app/env`.

Configuring External Resources

When accessing external resources (resources not deployed with the application EAR file) such as a DataSource from a Web application via Java Naming and Directory Interface (JNDI), you can map the JNDI name you look up in your code to the actual JNDI name as bound in the global JNDI tree. This mapping is made using both the `web.xml` and `weblogic.xml` deployment descriptors and allows you to change these resources without changing your application code. You provide a name that is used in your Java code, the name of the resource as bound in the JNDI tree, and the Java type of the resource, and you indicate whether security for the resource is handled programmatically by the servlet or from the credentials associated with the HTTP request.

To configure external resources:

1. Enter the resource name in the deployment descriptor as you use it in your code, the Java type, and the security authorization type.
2. Map the resource name to the JNDI name.

The following example illustrates how to use an external DataSource. It assumes that you have defined a data source called `accountDataSource`. For more information, see [JDBC Data Sources Online Help](#).

Listing 3-5 Using an External DataSource

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>
</resource-description>
```

Referencing External EJBs

Web applications can access EJBs that are deployed as part of a different application (a different EAR file) by using an external reference. The EJB being referenced exports a name to the global JNDI tree in its `weblogic-ejb-jar.xml` deployment descriptor. An EJB reference in the Web application module can be linked to this global JNDI name by adding an

`<ejb-reference-description>` element to its `weblogic.xml` deployment descriptor.

This procedure provides a level of indirection between the Web application and an EJB and is useful if you are using third-party EJBs or Web applications and cannot modify the code to directly call an EJB. In most situations, you can call the EJB directly without using this indirection. For more information, see [Programming WebLogic Enterprise JavaBeans](#).

To reference an external EJB for use in a Web application:

1. Enter the EJB reference name you use to look up the EJB in your code, the Java class name and the class name of the home and remote interfaces of the EJB in the `<ejb-ref>` element of the Web Application deployment descriptor.
2. Map the reference name in the `<ejb-reference-description>` element of the WebLogic-specific deployment descriptor, `weblogic.xml` to the JNDI name defined in the `weblogic-ejb-jar.xml` file.

If the Web application is part of an Enterprise Application Archive (EAR file), you can reference an EJB by the name used in the EAR with the `<ejb-link>` element.

More about the `ejb-ref*` Elements

The `ejb-ref` element in the `web.xml` deployment descriptor declares that either a servlet or JSP is going to be using a particular EJB. The `ejb-reference-description` element in the `weblogic.xml` deployment descriptor binds that reference to an EJB, which is advertised in the global JNDI tree.

The `ejb-reference-descriptor` element indicates which `ejb-ref` element it is resolving with the `ejb-ref-name` element. That is, the `ejb-reference-descriptor` and `ejb-ref` elements with the same `ejb-ref-name` element go together.

With the addition of the `ejb-link` syntax, the `ejb-reference-descriptor` element is no longer required if the EJB being used is in the same application as the servlet or JSP that is using the EJB.

The `ejb-ref-name` element serves two purposes in the `web.xml` deployment descriptor:

- It is the name that the user code (servlet or JSP) uses to look up the EJB. Therefore, if your `ejb-ref-name` element is `ejb1`, you would perform a JNDI name lookup for `ejb1` relative to `java:comp/env`. The `ejb-ref-name` element is bound into the component environment (`java:comp/env`) of the Web application containing the servlet or JSP.

Assuming the `ejb-ref-name` element is `ejb1`, the code in your servlet or JSP should look like:

```
Context ctx = new InitialContext();
ctx = (Context)ctx.lookup("java:comp/env");
Object o = ctx.lookup("ejb1");
Ejb1Home home = (Ejb1Home) PortableRemoteObject.narrow(o,
Ejb1Home.class);
```

- It links the `ejb-ref` and `ejb-reference-descriptor` elements together.

Referencing Application-Scoped EJBs

Within an application, WebLogic Server binds any EJBs referenced by other application components to the environments associated with those referencing components. These resources are accessed at runtime through a JNDI name lookup relative to `java:comp/env`.

The following is an example of an application deployment descriptor (`application.xml`) for an application containing an EJB and a Web application, also called an Enterprise Application. (For the sake of brevity, the XML header is not included in this example.)

Listing 3-6 Example Deployment Descriptor

```
<application>
  <display-name>MyApp</display-name>
  <module>
    <web>
      <web-uri>myapp.war</web-uri>
      <context-root>myapp</context-root>
    </web>
  </module>
  <module>
    <ejb>ejb1.jar</ejb>
  </module>
</application>
```

To allow the code in the Web application to use an EJB in `ejb1.jar`, the Web application deployment descriptor (`web.xml`) must include an `<ejb-ref>` stanza that contains an `<ejb-link>` referencing the JAR file and the name of the EJB that is being called.

The format of the `<ejb-link>` entry must be as follows:

```
filename#ejbname
```

where `filename` is the name of the JAR file, relative to the Web application, and `ejbname` is the EJB within that JAR file. The `<ejb-link>` element should look like the following:

```
<ejb-link>../ejb1.jar#myejb</ejb-link>
```

Note that since the JAR path is relative to the WAR file, it begins with `"/`. Also, if the `ejbname` is unique across the application, the JAR path may be dropped. As a result, your entry may look like the following:

```
<ejb-link>myejb</ejb-link>
```

The `<ejb-link>` element is a sub-element of an `<ejb-ref>` element contained in the Web application's `web.xml` descriptor. The `<ejb-ref>` element should look like the following:

Listing 3-7 <ejb-ref> Element

```

<web-app>
  ...
  <ejb-ref>
    <ejb-ref-name>ejb1</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>mypackage.ejb1.MyHome</home>
    <remote>mypackage.ejb1.MyRemote</remote>
    <ejb-link>../ejb1.jar#myejb</ejb-link>
  </ejb-ref>
  ...
</web-app>

```

Referring to the syntax for the `<ejb-link>` element in the above example,

```
<ejb-link>../ejb1.jar#ejb1</ejb-link>
```

the portion of the syntax to the left of the # is a relative path to the EJB module being referenced. The syntax to the right of # is the particular EJB being referenced in that module. In the above example, the EJB JAR and WAR files are at the same level.

The name referenced in the `<ejb-link>` (in this example, `myejb`) corresponds to the `<ejb-name>` element of the referenced EJB's descriptor. As a result, the deployment descriptor (`ejb-jar.xml`) of the EJB module that this `<ejb-ref>` is referencing should have an entry similar to the following:

Listing 3-8 <ejb-jar> Element

```

<ejb-jar>
  ...
  <enterprise-beans>

```

```
<session>
  <ejb-name>myejb</ejb-name>
  <home>mypackage.ejb1.MyHome</home>
  <remote>mypackage.ejb1.MyRemote</remote>
  <ejb-class>mypackage.ejb1.MyBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
...
</ejb-jar>
```

Notice the `<ejb-name>` element is set to `myejb`.

At runtime, the Web application code looks up the EJB's JNDI name relative to `java:/comp/env`. The following is an example of the servlet code:

```
MyHome home = (MyHome)ctx.lookup("java:/comp/env/ejb1");
```

The name used in this example (`ejb1`) is the `<ejb-ref-name>` defined in the `<ejb-ref>` element of the `web.xml` segment above.

Loading Servlets, Context Listeners, and Filters

Servlets, Context Listeners, and Filters are loaded and destroyed in the following order:

Order of loading:

1. Context Listeners
2. Filters
3. Servlets

Order of destruction:

1. Servlets

2. Filters
3. Context Listeners

Servlets and filters are loaded in the same order they are defined in the `web.xml` file and unloaded in reverse order. Context listeners are loaded in the following order:

1. All context listeners in the `web.xml` file in the order as specified in the file
2. Packaged JAR files containing tag library descriptors
3. Tag library descriptors in the WEB-INF directory

Determining the Encoding of an HTTP Request

WebLogic Server needs to convert character data contained in an HTTP request from its native encoding to the Unicode encoding used by the Java servlet API. In order to perform this conversion, WebLogic Server needs to know which code set was used to encode the request.

There are two ways you can define the code set:

- For a POST operation, you can set the encoding in the HTML `<form>` tag. For example, this form tag sets `SJIS` as the character set for the content:

```
<form action="http://some.host.com/myWebApp/foo/index.html">
  <input type="application/x-www-form-urlencoded; charset=SJIS">
</form>
```

When the form is read by WebLogic Server, it processes the data using the `SJIS` character set.

- Because all Web clients do not transmit the information after the semicolon in the above example, you can set the code set to be used for requests by using the `<input-charset>` element in the WebLogic-specific deployment descriptor, `weblogic.xml`. The `<java-charset-name>` element defines the encoding used to convert data when the URL of the request contains the path specified with the `<resource-path>` element.

For example:

```
<input-charset>
<resource-path>/foo/*</resource-path>
<java-charset-name>SJIS</java-charset-name>
</input-charset>
```

This method works for both `GET` and `POST` operations.

Mapping IANA Character Sets to Java Character Sets

The names assigned by the Internet Assigned Numbers Authority (IANA) to describe character sets are sometimes different from the names used by Java. Because all HTTP communication uses the IANA character set names and these names are not always the same, WebLogic Server internally maps IANA character set names to Java character set names and can usually determine the correct mapping. However, you can resolve any ambiguities by explicitly mapping an IANA character set to the name of a Java character set.

To map an IANA character set to a Java character, set the character set names in the `<charset-mapping>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. Define the IANA character set name in the `<iana-charset-name>` element and the Java character set name in the `<java-charset-name>` element. For example:

```
<charset-mapping>
  <iana-charset-name>Shift-JIS</iana-charset-name>
  <java-charset-name>SJIS</java-charset-name>
</charset-mapping>
```

Using Sessions and Session Persistence in Web Applications

The following sections describe how to set up sessions and session persistence:

- [“Overview of HTTP Sessions” on page 4-1](#)
- [“Setting Up Session Management” on page 4-1](#)
- [“Configuring Session Persistence” on page 4-4](#)
- [“Using URL Rewriting Instead of Cookies” on page 4-10](#)

Overview of HTTP Sessions

Session tracking enables you to track a user's progress over multiple servlets or HTML pages, which, by nature, are stateless. A *session* is defined as a series of related browser requests that come from the same client during a certain time period. Session tracking ties together a series of browser requests—think of these requests as pages—that may have some meaning as a whole, such as a shopping cart application.

Setting Up Session Management

WebLogic Server is set up to handle session tracking by default. You need not set any of these properties to use session tracking. However, configuring how WebLogic Server manages sessions is a key part of tuning your application for best performance. When you set up session management, you determine factors such as:

- How many users you expect to hit the servlet

- How long each session lasts
- How much data you expect to store for each user
- Heap size allocated to the WebLogic Server instance

You can also store data permanently from an HTTP session. See [“Configuring Session Persistence” on page 4-4](#).

For information on editing deployment descriptors, see [“Deployment Descriptors” on page 2-2](#) in [Chapter 2, “Deploying Web Applications.”](#)

HTTP Session Properties

You configure WebLogic Server session tracking by defining properties in the WebLogic-specific deployment descriptor, `weblogic.xml`. For a complete list of session attributes, see [“session-descriptor” on page B-5](#).

WebLogic Server 7.0 introduced a change to the SessionID format that caused some load balancers to lose the ability to retain session stickiness.

A new server startup flag, `-Dweblogic.servlet.useExtendedSessionFormat=true`, retains the information that the load-balancing application needs for session stickiness. The extended session ID format will be part of the URL if URL rewriting is activated, and the startup flag is set to true.

Session Timeout

You can specify an interval of time after which HTTP sessions expire. When a session expires, all data stored in the session is discarded. You can set the interval in either `web.xml` or `weblogic.xml`:

- Set the `TimeoutSecs` attribute in the `session-descriptor` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. This value is set in seconds. For more information, see [“session-descriptor” on page B-5](#).
- Set the `session-timeout` element in the Web Application deployment descriptor, `web.xml`. For more information, see [“session-config” on page A-11](#).

Configuring WebLogic Server Session Cookies

WebLogic Server uses cookies for session management when cookies are supported by the client browser.

The cookies that WebLogic Server uses to track sessions are set as transient by default and do not outlive the session. When a user quits the browser, the cookies are lost and the session ends. This behavior is in the spirit of session usage and it is recommended that you use sessions in this way.

You can configure session-tracking attributes of cookies in the WebLogic-specific deployment descriptor, `weblogic.xml`. A complete list of session and cookie-related attributes is available in [“session-descriptor” on page B-5](#).

Configuring Application Cookies That Outlive a Session

For longer-lived client-side user data, you program your application to create and set its own cookies on the browser via the HTTP servlet API. The application should not attempt to use the cookies associated with the HTTP session. Your application might use cookies to auto-login a user from a particular machine, in which case you would set a new cookie to last for a long time. Remember that the cookie can only be sent from that particular client machine. Your application should store data on the server if it must be accessed by the user from multiple locations.

You cannot directly connect the age of a browser cookie with the length of a session. If a cookie expires before its associated session, that session becomes orphaned. If a session expires before its associated cookie, the servlet is not be able to find a session. At that point, a new session is automatically assigned when the `request.getSession(true)` method is called.

You can set the maximum life of a cookie with the `CookieMaxAgeSecs` attribute in the session descriptor of the `weblogic.xml` deployment descriptor.

Logging Out and Ending a Session

User authentication information is stored both in the user's session data and in the context of a server or virtual host that is targeted by a Web application. The `session.invalidate()` method, which is often used to log out a user, only invalidates the current session for a user—the user's authentication information still remains valid and is stored in the context of the server or virtual host. If the server or virtual host is hosting only one Web application, the `session.invalidate()` method, in effect, logs out the user.

There are several Java methods and strategies you can use when using authentication with multiple Web applications. For more information, see ["Implementing Single Sign-On" in *Programming WebLogic HTTP Servlets*](#).

Configuring Session Persistence

You use session persistence to permanently store data from an HTTP session object to enable failover and load balancing across a cluster of WebLogic Servers. When your applications stores data in an HTTP session object, the data must be serializable.

There are five different implementations of session persistence:

- Memory (single-server, non-replicated)
- File system persistence
- JDBC persistence
- Cookie-based session persistence
- In-memory replication (across a cluster)

The first four are discussed here; in-memory replication is discussed in “[HTTP Session State Replication](#),” in *Using WebLogic Server Clusters*.

File, JDBC, cookie-based, and memory (single-server, non-populated) session persistence have some common properties. Each persistence method has its own set of attributes, as discussed in the following sections. These attributes are part of the `session-param` element, which is a child element of the `session-descriptor` element in the `weblogic.xml` deployment descriptor file.

Attributes Shared by Different Types of Session Persistence

This section describes attributes common to file and JDBC-based persistence. You can configure the number of sessions that are held in memory by defining the following attributes in the `<session-param>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. These properties are only applicable if you are using session persistence:

CacheSize

Limits the number of cached sessions that can be active in memory at any one time. If you expect high volumes of simultaneous active sessions, you do not want these sessions to soak up the RAM of your server because this may cause performance problems swapping to and from virtual memory. When the cache is full, the least recently used sessions are stored in the persistent store and recalled automatically when required. If you do not use persistence, this property is ignored, and there is no soft limit to the number of sessions allowed in main memory. By default, the number of cached sessions is 256. To turn off caching, set this to 0.

Note: `CacheSize` is used by JDBC and file-based sessions only for maintaining the in-memory bubbling cache. It is not applicable for other persistence types.

`InvalidationIntervalSecs`

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. Set this attribute to a value less than the value set for the `<session-timeout>` element. Use this attribute 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 not set, the attribute defaults to 60 seconds.

To set `<session-timeout>`, see [“session-config” on page A-11](#).

Using Memory-based, Single-server, Non-replicated Persistent Storage

When you use memory-based storage, all session information is stored in memory and is lost when you stop and restart WebLogic Server. To use memory-based, single-server, non-replicated persistent storage, set the `PersistentStoreType` attribute in the `<session-param>` element of `weblogic.xml` file to `memory`.

Note: If you do not allocate sufficient heap size when running WebLogic Server, your server may run out of memory under heavy load.

Using File-based Persistent Storage

To configure file-based persistent storage for sessions:

1. In the deployment descriptor file `weblogic.xml`, set the `PersistentStoreType` attribute in the `<session-param>` element to `file`.
2. Set the directory where WebLogic Server stores the sessions. See [“PersistentStoreDir” on page B-9](#).

Note: You must create this directory yourself and make sure appropriate access privileges have been assigned to the directory.

Using a Database for Persistent Storage (JDBC persistence)

JDBC persistence stores session data in a database table using a schema provided for this purpose. You can use any database for which you have a JDBC driver. You configure database access by using connection pools.

Because WebLogic Server uses the system time to determine the session lifetime when using JDBC session persistence, you must be sure to synchronize the system clock on all of the machines on which servers are running in the same cluster.

To configure JDBC-based persistent storage for sessions:

1. In `weblogic.xml`, set the `PersistentStoreType` attribute in the `<session-param>` element to `jdbc`.
2. Set a JDBC connection pool to be used for persistence storage with the `PersistentStorePool` property in the `<session-descriptor>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. Use the name of a connection pool that is defined in the WebLogic Server Administration Console.
3. Set up a database table named `wl_servlet_sessions` for JDBC-based persistence. The connection pool that connects to the database needs to have read/write access for this table.

Note: Create indexes on `wl_id` and `wl_context_path`, if the database does not create them automatically. Some databases create indexes automatically for primary keys.

Set up column names and data types as follows.

Table 4-1 Creating `wl_servlet_sessions` table

Column Name	Data Type
<code>wl_id</code>	Variable-width alphanumeric column, up to 100 characters; for example, Oracle <code>VARCHAR2(100)</code> . <i>The primary key must be set as follows:</i> <code>wl_id + wl_context_path</code> .
<code>wl_context_path</code>	Variable-width alphanumeric column, up to 100 characters; for example, Oracle <code>VARCHAR2(100)</code> . <i>This column is used as part of the primary key. (See the <code>wl_id</code> column description.)</i>
<code>wl_is_new</code>	Single char column; for example, Oracle <code>CHAR(1)</code>

Column Name	Data Type
wl_create_time	Numeric column, 20 digits; for example, Oracle NUMBER (20)
wl_is_valid	Single char column; for example, Oracle CHAR (1)
wl_session_values	Large binary column; for example, Oracle LONG RAW
wl_access_time	Numeric column, 20 digits; for example, NUMBER (20)
wl_max_inactive_interval	Integer column; for example, Oracle Integer. Number of seconds between client requests before the session is invalidated. A negative time value indicates that the session should never time out.

If you are using an Oracle DBMS, use the following SQL statement to create the `wl_servlet_sessions` table. Modify the SQL statement for use with your DBMS.

Listing 4-1 Creating `wl_servlet_sessions` table with Oracle DBMS

```
create table wl_servlet_sessions
( wl_id VARCHAR2(100) NOT NULL,
  wl_context_path VARCHAR2(100) NOT NULL,
  wl_is_new CHAR(1),
  wl_create_time NUMBER(20),
  wl_is_valid CHAR(1),
  wl_session_values LONG RAW,
  wl_access_time NUMBER(20),
  wl_max_inactive_interval INTEGER,
  PRIMARY KEY (wl_id, wl_context_path) );
```

Note: You can use the `JDBCConnectionTimeoutSecs` attribute to configure a maximum duration that JDBC session persistence should wait for a JDBC connection from the connection pool before failing to load the session data. For more information, see [“JDBCConnectionTimeoutSecs” on page B-9](#).

If you are using SqlServer2000, use the following SQL statement to create the `wl_servlet_sessions` table. Modify the SQL statement for use with your DBMS.

Listing 4-2 Creating `wl_servlet_sessions` table with SqlServer 2000

```
create table wl_servlet_sessions
( wl_id VARCHAR2(100) NOT NULL,
  wl_context_path VARCHAR2(100) NOT NULL,
  wl_is_new VARCHAR(1),
  wl_create_time DECIMAL,
  wl_is_valid VARCHAR(1),
  wl_session_values IMAGE,
  wl_access_time DECIMAL,
  wl_max_inactive_interval INTEGER,
  PRIMARY KEY (wl_id, wl_context_path) );
```

If you are using Pointbase, Pointbase translates the SQL. For example, Pointbase would translate the SQL provided in [Listing 4-1](#) as follows.

Listing 4-3 Creating `wl_servlet_sessions` table with Pointbase SQL Translation

```
SQL> describe wl_servlet_sessions;
WL_SERVLET_SESSIONS
WL_ID VARCHAR(100) NULLABLE: NO
WL_CONTEXT_PATH VARCHAR(100) NULLABLE: NO
WL_IS_NEW CHARACTER(1) NULLABLE: YES
WL_CREATE_TIME DECIMAL(20) NULLABLE: YES
WL_IS_VALID CHARACTER(1) NULLABLE: YES
WL_SESSION_VALUES BLOB(65535) NULLABLE: YES
```

WL_ACCESS_TIME DECIMAL(20) NULLABLE: YES

WL_MAX_INACTIVE_INTERVAL INTEGER(10) NULLABLE: YES

Primary Key: WL_CONTEXT_PATH

Primary Key: WL_ID

Using Cookie-Based Session Persistence

Cookie-based session persistence provides a stateless solution for session persistence by storing all session data in a cookie in the user's browser. Cookie-based session persistence is most useful when you do not need to store large amounts of data in the session. Cookie-based session persistence can make managing your WebLogic Server installation easier because clustering failover logic is not required. Because the session is stored in the browser, not on the server, you can start and stop WebLogic Servers without losing sessions.

There are some limitations to cookie-based session persistence:

- You can store only string attributes in the session. If you store any other type of object in the session, an `IllegalArgumentException` exception is thrown.
- You cannot flush the HTTP response (because the cookie must be written to the header data *before* the response is committed).
- If the content length of the response exceeds the buffer size, the response is automatically flushed and the session data cannot be updated in the cookie. (The buffer size is, by default, 8192 bytes. You can change the buffer size with the `javax.servlet.ServletResponse.setBufferSize()` method.)
- You can only use basic (browser-based) authentication.
- Session data is sent to the browser in clear text.
- The user's browser must be configured to accept cookies.
- You cannot use commas (,) in a string when using cookie-based session persistence or an exception occurs.

To set up cookie-based session persistence:

1. In the `<session-param>` element of `weblogic.xml`, set the `PersistentStoreType` attribute to `cookie`.

2. Optionally, set a name for the cookie using the `PersistentStoreCookieName` attribute. The default is `WLCookie`.

Using URL Rewriting Instead of Cookies

In some situations, a browser or wireless device may not accept cookies, which makes session tracking with cookies impossible. URL rewriting is a solution to this situation that can be substituted automatically when WebLogic Server detects that the browser does not accept cookies. URL rewriting involves encoding the session ID into the hyper-links on the Web pages that your servlet sends back to the browser. When the user subsequently clicks these links, WebLogic Server extracts the ID from the URL address and finds the appropriate `HttpSession` when your servlet calls the `getSession()` method.

Enable URL rewriting in WebLogic Server by setting the `URLRewritingEnabled` attribute in the WebLogic-specific deployment descriptor, `weblogic.xml`, under the `<session-param>` element. The default value for this attribute is `true`. See “[URLRewritingEnabled](#)” on page B-11.

Coding Guidelines for URL Rewriting

Here are general guidelines for supporting URL rewriting.

- Avoid writing a URL straight to the output stream, as shown here:

```
out.println("<a href=\"/myshop/catalog.jsp\">catalog</a>");
```

Instead, use the `HttpServletResponse.encodeURL()` method, for example:

```
out.println("<a href=\""
    + response.encodeURL("myshop/catalog.jsp")
    + "\">catalog</a>");
```

Calling the `encodeURL()` method determines whether the URL needs to be rewritten. If it does need to be rewritten, WebLogic Server rewrites the URL by appending the session ID to the URL, with the session ID preceded by a semicolon.

- In addition to URLs that are returned as a response to WebLogic Server, also encode URLs that send redirects. For example:

```
if (session.isNew())
    response.sendRedirect (response.encodeRedirectUrl(welcomeURL));
```

WebLogic Server uses URL rewriting when a session is new, even if the browser does accept cookies, because the server cannot tell whether a browser accepts cookies in the first visit of a session.

- Your servlet can determine whether a given session ID was received from a cookie by checking the Boolean returned from the `HttpServletRequest.isRequestedSessionIdFromCookie()` method. Your application may respond appropriately, or simply rely on URL rewriting by WebLogic Server.

URL Rewriting and Wireless Access Protocol (WAP)

If you are writing a WAP application, you *must* use URL rewriting because the WAP protocol does not support cookies.

In addition, some WAP devices have a 128-character limit on the length of a URL (including attributes), which limits the amount of data that can be transmitted using URL rewriting. To allow more space for attributes, you can limit the size of the session ID that is randomly generated by WebLogic Server.

In particular, to use the `WAPEnabled` attribute, use the Administration Console at Server → Protocols → HTTP → Advanced Options. The `WAPEnabled` attribute restricts the size of the session ID to 52 characters and disallows special characters, such as ! and #. You can also use the `IDLength` parameter of `weblogic.xml` to further restrict the size of the session ID. For additional details, see [“IDLength” on page B-8](#).

Using Sessions and Session Persistence in Web Applications

Application Events and Event Listener Classes

The following sections discuss application events and event listener classes:

- [“Overview of Application Event Listener Classes”](#) on page 5-2
- [“Servlet Context Events”](#) on page 5-2
- [“HTTP Session Events”](#) on page 5-3
- [“Configuring an Event Listener Class”](#) on page 5-4
- [“Writing an Event Listener Class”](#) on page 5-5
- [“Templates for Event Listener Classes”](#) on page 5-5
- [“Additional Resources”](#) on page 5-7

Overview of Application Event Listener Classes

Application events provide notifications of a change in state of the *servlet context* (each Web application uses its own servlet context) or of an *HTTP session object*. You write event listener classes that respond to these changes in state, and you configure and deploy them in a Web application. The servlet container generates events that cause the event listener classes to do something. In other words, the servlet container calls the methods on a user's event listener class.

The following is an overview of this process:

1. The user creates an event listener class that implements one of the listener interfaces.
2. This implementation is registered in the deployment descriptor.
3. At deployment time, the servlet container constructs an instance of the event listener class. (This is why the public constructor must exist, as discussed in [“Writing an Event Listener Class” on page 5-5.](#))
4. At runtime, the servlet container invokes on the instance of the the listener class.

For servlet context events, the event listener classes can receive notification when the Web application is deployed or undeployed (or when WebLogic Server shuts down), and when attributes are added, removed, or replaced.

For HTTP session events, the event listener classes can receive notification when an HTTP session is activated or is about to be passivated, and when an HTTP session attribute is added, removed, or replaced.

Use Web application event listener classes to:

- Manage database connections when a Web application is deployed or shuts down
- Create standard counter utilities
- Monitor the state of HTTP sessions and their attributes

Servlet Context Events

The following table lists the types of Servlet context events, the interface your event listener class must implement to respond to each Servlet context event, and the methods invoked when the Servlet context event occurs.

Table 5-1 Servlet Context Events

Type of Event	Interface	Method
Servlet context is created.	<code>javax.servlet.ServletContextListener</code>	<code>contextInitialized()</code>
Servlet context is about to be shut down.	<code>javax.servlet.ServletContextListener</code>	<code>contextDestroyed()</code>
An attribute is added.	<code>javax.servlet.ServletContextAttributesListener</code>	<code>attributeAdded()</code>
An attribute is removed.	<code>javax.servlet.ServletContextAttributesListener</code>	<code>attributeRemoved()</code>
An attribute is replaced.	<code>javax.servlet.ServletContextAttributesListener</code>	<code>attributeReplaced()</code>

HTTP Session Events

The following table lists the types of HTTP session events your event listener class must implement to respond to the HTTP session events and the methods invoked when the HTTP session events occur.

Table 5-2 HTTP Session Events

Type of Event	Interface	Method
An HTTP session is activated.	<code>javax.servlet.http.HttpSessionListener</code>	<code>sessionCreated()</code>
An HTTP session is about to be passivated.	<code>javax.servlet.http.HttpSessionListener</code>	<code>sessionDestroyed()</code>
An attribute is added.	<code>javax.servlet.http.HttpSessionAttributeListener</code>	<code>attributeAdded()</code>

Type of Event	Interface	Method
An attribute is removed.	<code>javax.servlet.http.HttpSessionAttributeListener</code>	<code>attributeRemoved()</code>
An attribute is replaced.	<code>javax.servlet.http.HttpSessionAttributeListener</code>	<code>attributeReplaced()</code>

Note: The Servlet 2.3 specification also contains the `javax.servlet.http.HttpSessionBindingListener` and the `javax.servlet.http.HttpSessionActivationListener` interfaces. These interfaces are implemented by objects that are stored as session attributes and do not require registration of an event listener in `web.xml`. For more information, see the Javadocs for these interfaces.

Configuring an Event Listener Class

To configure an event listener class:

1. Open the `web.xml` deployment descriptor of the Web application for which you are creating an event listener class in a text editor. The `web.xml` file is located in the `WEB-INF` directory of your Web application.
2. Add an event declaration using the `<listener>` element. The event declaration defines the event listener class that is invoked when the event occurs. The `<listener>` element must directly follow the `<filter>` and `<filter-mapping>` elements and directly precede the `<servlet>` element. You can specify more than one event listener class for each type of event. WebLogic Server invokes the event listener classes in the order that they appear in the deployment descriptor (except for shutdown events, which are invoked in the reverse order). For example:

```
<listener>
  <listener-class>myApp.MyContextListenerClass</listener-class>
</listener>

<listener>
  <listener-class>myApp.MySessionAttributeListenerClass</listener-class>
>
</listener>
```

3. Write and deploy the event listener class. For details, see the section, [“Writing an Event Listener Class” on page 5-5](#).

Writing an Event Listener Class

To write an event listener class:

1. Create a new event listener class that implements the appropriate interface for the type of event to which your class responds. For a list of these interfaces, see “[Servlet Context Events](#)” on page 5-2 or “[HTTP Session Events](#)” on page 5-3. See “[Templates for Event Listener Classes](#)” on page 5-5 for sample templates you can use to get started.

2. Create a public constructor that takes no arguments. For example:

```
public class MyListener {
    // public constructor
    public MyListener() { /* ... */ }
}
```

3. Implement the required methods of the interface. See the [J2EE API Reference \(Javadocs\)](http://java.sun.com/j2ee/tutorial/api/index.html) at <http://java.sun.com/j2ee/tutorial/api/index.html> for more information.
4. Copy the compiled event listener classes into the `WEB-INF/classes` directory of the Web application, or package them into a JAR file and copy the JAR file into the `WEB-INF/lib` directory of the Web application.

The following useful classes are passed into the methods in an event listener class:

```
javax.servlet.http.HttpSessionEvent
    provides access to the HTTP session object

javax.servlet.ServletContextEvent
    provides access to the servlet context object.

javax.servlet.ServletContextAttributeEvent
    provides access to servlet context and its attributes

javax.servlet.http.HttpSessionBindingEvent
    provides access to an HTTP session and its attributes
```

Templates for Event Listener Classes

The following examples provide some basic templates for event listener classes.

Servlet Context Event Listener Class Example

```
package myApp;
import javax.servlet.http.*;
```

Application Events and Event Listener Classes

```
public final class MyContextListenerClass implements
ServletContextListener {
    public void contextInitialized(ServletContextEvent event) {

        /* This method is called when the servlet context is
           initialized(when the Web application is deployed).
           You can initialize servlet context related data here.
        */

    }

    public void contextDestroyed(ServletContextEvent event) {

        /* This method is invoked when the Servlet Context
           (the Web application) is undeployed or
           WebLogic Server shuts down.
        */

    }
}
```

HTTP Session Attribute Event Listener Class Example

```
package myApp;
import javax.servlet.*;

public final class MySessionAttributeListenerClass implements
HttpSessionAttributeListener {

    public void attributeAdded(HttpSessionBindingEvent sbe) {
        /* This method is called when an attribute
           is added to a session.
        */
    }

    public void attributeRemoved(HttpSessionBindingEvent sbe) {
        /* This method is called when an attribute
           is removed from a session.
        */
    }
}
```

```
    */  
}  
  
public void attributeReplaced(HttpSessionBindingEvent sbe) {  
    /* This method is invoked when an attribute  
       is replaced in a session.  
    */  
}  
}
```

Additional Resources

- [Servlet 2.3 Specification from Sun Microsystems](http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html) at <http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html>
- [J2EE API Reference \(Javadocs\)](http://java.sun.com/j2ee/tutorial/api/index.html) at <http://java.sun.com/j2ee/tutorial/api/index.html>
- [The J2EE Tutorial](http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html) from Sun Microsystems: at http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html

Application Events and Event Listener Classes

Filters

The following sections provide information about using filters in a Web application:

- [“Overview of Filters” on page 6-1](#)
- [“Writing a Filter Class” on page 6-2](#)
- [“Configuring Filters” on page 6-3](#)
- [“Filtering the Servlet Response Object” on page 6-5](#)
- [“Additional Resources” on page 6-6](#)

Overview of Filters

A filter is a Java class that is invoked in response to a request for a resource in a Web application. Resources include Java Servlets, JavaServer pages (JSP), and static resources such as HTML pages or images. A filter intercepts the request and can examine and modify the response and request objects or execute other tasks.

Filters are an advanced J2EE feature primarily intended for situations where the developer cannot change the coding of an existing resource and needs to modify the behavior of that resource. Generally, it is more efficient to modify the code to change the behavior of the resource itself rather than using filters to modify the resource. In some situations, using filters can add unnecessary complexity to an application and degrade performance.

How Filters Work

You define filters in the context of a Web application. A filter intercepts a request for a specific named resource or a group of resources (based on a URL pattern) and executes the code in the filter. For each resource or group of resources, you can specify a single filter or multiple filters that are invoked in a specific order, called a *chain*.

When a filter intercepts a request, it has access to the `javax.servlet.HttpServletRequest` and `javax.servlet.HttpServletResponse` objects that provide access to the HTTP request and response, and a `javax.servlet.FilterChain` object. The `FilterChain` object contains a list of filters that can be invoked sequentially. When a filter has completed its work, the filter can either call the next filter in the chain, block the request, throw an exception, or invoke the originally requested resource.

After the original resource is invoked, control is passed back to the filter at the bottom of the list in the chain. This filter can then examine and modify the response headers and data, block the request, throw an exception, or invoke the next filter up from the bottom of the chain. This process continues in reverse order up through the chain of filters.

Note: The filter can modify the headers only if the response has not already been committed.

Uses for Filters

Filters can be useful for the following functions:

- Implementing a logging function
- Implementing user-written security functionality
- Debugging
- Encryption
- Data compression
- Modifying the response sent to the client. (However, post processing the response can degrade the performance of your application.)

Writing a Filter Class

To write a filter class, implement the `javax.servlet.Filter` interface (see <http://java.sun.com/j2ee/tutorial/api/javax/servlet/Filter.html>). You must implement the following methods of this interface:

- `init()`
- `destroy()`
- `doFilter()`

You use the `doFilter()` method to examine and modify the request and response objects, perform other tasks such as logging, invoke the next filter in the chain, or block further processing.

Several other methods are available on the `FilterConfig` object for accessing the name of the filter, the `ServletContext` and the filter's initialization attributes. For more information see the [J2EE Javadocs](#) from Sun Microsystems for `javax.servlet.FilterConfig`. Javadocs are available at <http://java.sun.com/j2ee/tutorial/api/index.html>.

To access the next item in the chain (either another filter or the original resource, if that is the next item in the chain), call the `FilterChain.doFilter()` method.

Configuring Filters

You configure filters as part of a Web application, using the application's `web.xml` deployment descriptor. In the deployment descriptor, you specify the filter and then map the filter to a URL pattern or to a specific servlet in the Web application. You can specify any number of filters.

Configuring a Filter

To configure a filter:

1. Open the `web.xml` deployment descriptor in a text editor or use the Administration Console. For more information, see [“Web Application Developer Tools” on page 1-8](#). The `web.xml` file is located in the `WEB-INF` directory of your Web application.
2. Add a filter declaration. The `<filter>` element declares a filter, defines a name for the filter, and specifies the Java class that executes the filter. The `<filter>` element must directly follow the `<context-param>` element and directly precede the `<listener>` and `<servlet>` elements. For example:

```
<context-param>Param</context-param>
<filter>
  <icon>
    <small-icon>MySmallIcon.gif</small-icon>
    <large-icon>MyLargeIcon.gif</large-icon>
  </icon>
  <filter-name>myFilter</filter-name>
```

```
<display-name>My Filter</display-name>
<description>This is my filter</description>
<filter-class>examples.myFilterClass</filter-class>
</filter>

<listener>Listener</listener>

<servlet>Servlet</servlet>
```

The icon, description, and display-name elements are optional.

3. Specify one or more initialization attributes inside a `<filter>` element. For example:

```
<filter>
  <icon>
    <small-icon>MySmallIcon.gif</small-icon>
    <large-icon>MyLargeIcon.gif</large-icon>
  </icon>
  <filter-name>myFilter</filter-name>
  <display-name>My Filter</display-name>
  <description>This is my filter</description>
  <filter-class>examples.myFilterClass</filter-class>
  <init-param>
    <param-name>myInitParam</param-name>
    <param-value>myInitParamValue</param-value>
  </init-param>
</filter>
```

Your Filter class can read the initialization attributes using the

`FilterConfig.getInitParameter()` or `FilterConfig.getInitParameters()` methods.

4. Add filter mappings. The `<filter-mapping>` element specifies which filter to execute based on a URL pattern or servlet name. The `<filter-mapping>` element must immediately follow the `<filter>` element(s).
 - To create a filter mapping using a URL pattern, specify the name of the filter and a URL pattern. URL pattern matching is performed according to the rules specified in the [Servlet 2.3 Specification from Sun Microsystems](http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html) at <http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html>, in section 11.1. For example, the following filter-mapping maps `myFilter` to requests that contain `/myPattern/`.

```
<filter-mapping>
  <filter-name>myFilter</filter-name>
  <url-pattern>/myPattern/*</url-pattern>
</filter-mapping>
```

- To create a filter mapping for a specific servlet, map the filter to the name of a servlet that is registered in the Web application. For example, the following code maps the `myFilter` filter to a servlet called `myServlet`:

```
<filter-mapping>
  <filter-name>myFilter</filter-name>
  <servlet-name>myServlet</servlet-name>
</filter-mapping>
```

5. To create a chain of filters, specify multiple filter mappings. For more information, see [“Configuring a Chain of Filters” on page 6-5](#).

Configuring a Chain of Filters

WebLogic Server creates a *chain* of filters by creating a list of all the filter mappings that match an incoming HTTP request. The ordering of the list is determined by the following sequence:

1. Filters where the `filter-mapping` element contains a `url-pattern` that matches the request are added to the chain in the order they appear in the `web.xml` deployment descriptor.
2. Filters where the `filter-mapping` element contains a `servlet-name` that matches the request are added to the chain *after* the filters that match a URL pattern.
3. The last item in the chain is always the originally requested resource.

In your filter class, use the `FilterChain.doFilter()` method to invoke the next item in the chain.

Filtering the Servlet Response Object

You can use filters to post-process the output of a servlet by appending data to the output generated by the servlet. However, in order to capture the output of the servlet, you must create a wrapper for the response. (You cannot use the original response object, because the output buffer of the servlet is automatically flushed and sent to the client when the servlet completes executing and *before* control is returned to the last filter in the chain.) When you create such a wrapper, WebLogic Server must manipulate an additional copy of the output in memory, which can degrade performance.

For more information on wrapping the response or request objects, see the [J2EE Javadocs](#) from Sun Microsystems for `javax.servlet.http.HttpServletResponseWrapper` and `javax.servlet.http.HttpServletRequestWrapper`. Javadocs are available at <http://java.sun.com/j2ee/tutorial/api/index.html>.

Additional Resources

- [Servlet 2.3 Specification from Sun Microsystems](http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html) at <http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html>
- [J2EE API Reference \(Javadocs\)](http://java.sun.com/j2ee/tutorial/api/index.html) at <http://java.sun.com/j2ee/tutorial/api/index.html>
- [The J2EE Tutorial](http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html) from Sun Microsystems at http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html

web.xml Deployment Descriptor Elements

The following sections describe the deployment descriptor elements defined in the `web.xml` file under the root element `<web-app>`:

- “context-param” on page A-4
- “description” on page A-3
- “display-name” on page A-2
- “distributable” on page A-3
- “ejb-ref” on page A-21
- “ejb-local-ref” on page A-22
- “env-entry” on page A-20
- “error-page” on page A-13
- “filter” on page A-5
- “filter-mapping” on page A-7
- “icon” on page A-2
- “listener” on page A-7
- “login-config” on page A-19
- “mime-mapping” on page A-12

- “resource-env-ref” on page A-14
- “resource-ref” on page A-15
- “security-constraint” on page A-16
- “security-role” on page A-20
- “servlet” on page A-8
- “servlet-mapping” on page A-10
- “session-config” on page A-11
- “taglib” on page A-13
- “welcome-file-list” on page A-12

icon

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.)

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

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

display-name

The optional `display-name` element specifies the Web application display name, a short name that can be displayed by GUI tools.

Element	Required/ Optional	Description
<display-name>	Optional	Currently, this element is not used by WebLogic Server.

description

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

Element	Required/ Optional	Description
<description>	Optional	Currently, this element is not used by WebLogic Server.

distributable

The distributable element is not used by WebLogic Server.

Element	Required/ Optional	Description
<distributable>	Optional	Currently, this element is not used by WebLogic Server.

context-param

The optional `context-param` element contains the declaration of a Web application's servlet context initialization parameters. The following table describes the reserved context parameters used by the Web application container, which have been deprecated and have replacements in `weblogic.xml`.

Deprecated Parameter	Description	Replacement Element in <code>weblogic.xml</code>
<code>weblogic.httpd.inputCharset</code>	Defines code set behavior for non-unicode operations.	<code>input-charset</code> (defined within <code>charset-param</code>) in <code>weblogic.xml</code> . See “input-charset” on page B-18 .
<code>weblogic.httpd.servlet.reloadCheckSecs</code>	Define how often WebLogic Server checks whether a servlet has been modified, and if so, reloads it. A value of -1 is never reload, 0 is always reload. The default is set to 1 second.	<code>servlet-reload-check-secs</code> (defined within <code>container-descriptor</code>) in <code>weblogic.xml</code> . See “container-descriptor” on page B-15 .
<code>weblogic.httpd.servlet.classpath</code>	When this value has been set, the container appends this path to the Web application classpath. This is not a recommended method and is supported only for backward compatibility.	No replacement. Use other means such as <code>manifest classpath</code> or <code>WEB-INF/lib</code> or <code>WEB-INF/classes</code> or virtual directories.
<code>weblogic.httpd.defaultServlet</code>	Sets the default servlet for the Web application. This is not a recommended method and is supported only for backward compatibility.	No replacement. Instead use the <code>servlet</code> and <code>servlet-mapping</code> elements in <code>web.xml</code> to define a default servlet. The URL pattern for <code>default-servlet</code> should be <code>/</code> . See “servlet-mapping” on page A-10 . For additional examples of servlet mapping, see “Servlet Mapping” on page 3-2 .

The following `context-param` parameter is still valid.

Element	Required/ Optional	Description
<code>weblogic.httpd. clientCertProxy</code>	optional	<p>This attribute specifies that certifications from clients of the Web application are provided in the special <code>WL-Proxy-Client-Cert</code> header sent by a proxy plug-in or <code>HttpClusterServlet</code>.</p> <p>This setting is useful if user authentication is performed on a proxy server—setting <code>clientCertProxy</code> causes the proxy server to pass on the certs to the cluster in a special header, <code>WL-Proxy-Client-Cert</code>.</p> <p>A <code>WL-Proxy-Client-Cert</code> header could be provided by any client with access to WebLogic Server. WebLogic Server takes the certificate information from that header, trusting that it came from a secure source (the plug-in) and uses that information to authenticate the user.</p> <p>For this reason, if you set <code>clientCertProxy</code>, use a connection filter to ensure that WebLogic Server accepts connections only from the machine on which the plug-in is running. See "Using Network Connection Filters" in <i>Programming WebLogic Security</i>.</p> <p>In addition to setting this attribute for an individual Web application, you can define this attribute:</p> <ul style="list-style-type: none"> • For all web applications hosted by a server instance, on the <code>Server->Configuration->General</code> page in the Administration Console • For all web applications hosted by server instances in a cluster, on the <code>Cluster->Configuration->General</code> page.

filter

The `filter` element defines a filter class and its initialization attributes. For more information on filters, see [“Configuring Filters” on page 6-3](#).

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

Element	Required/ Optional	Description
<code><icon></code>	Optional	Specifies the location within the Web application for a small and large image used to represent the filter in a GUI tool. Contains a <code>small-icon</code> and <code>large-icon</code> element. Currently, this element is not used by WebLogic Server.
<code><filter-name></code>	Required	Defines the name of the filter, used to reference the filter definition elsewhere in the deployment descriptor.
<code><display-name></code>	Optional	A short name intended to be displayed by GUI tools.
<code><description></code>	Optional	A text description of the filter.
<code><filter-class></code>	Required	The fully-qualified class name of the filter.
<code><init-param></code>	Optional	Contains a name/value pair as an initialization attribute of the filter. Use a separate set of <code><init-param></code> tags for each attribute.

filter-mapping

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

Element	Required/ Optional	Description
<code><filter-name></code>	Required	The name of the filter to which you are mapping a URL pattern or servlet. This name corresponds to the name assigned in the <code><filter></code> element with the <code><filter-name></code> element.
<code><url-pattern></code>	Required - or map by <code><servlet></code>	<p>Describes a pattern used to resolve URLs. The portion of the URL after the <code>http://host:port + ContextPath</code> is compared to the <code><url-pattern></code> by WebLogic Server. If the patterns match, the filter mapped in this element is called.</p> <p>Example patterns:</p> <pre>/soda/grape/* /foo/* /contents *.foo</pre> <p>The URL must follow the rules specified in the Servlet 2.3 Specification.</p>
<code><servlet-name></code>	Required - or map by <code><url-pattern></code>	The name of a servlet which, if called, causes this filter to execute.

listener

Define an application listener using the `listener` element.

Element	Required/ Optional	Description
<code><listener-class></code>	Optional	Name of the class that responds to a Web application event.

For more information, see [“Configuring an Event Listener Class” on page 5-4](#).

servlet

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 is precompiled and loaded when WebLogic Server starts.

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

Element	Required/ Optional	Description
<code><icon></code>	Optional	Location within the Web application for a small and large image used to represent the servlet in a GUI tool. Contains a <code>small-icon</code> and <code>large-icon</code> element. Currently, this element is not used by WebLogic Server.
<code><servlet-name></code>	Required	Defines the canonical name of the servlet, used to reference the servlet definition elsewhere in the deployment descriptor.
<code><display-name></code>	Optional	A short name intended to be displayed by GUI tools.
<code><description></code>	Optional	A text description of the servlet.
<code><servlet-class></code>	Required (or use <code><jsp-file></code>)	The fully-qualified class name of the servlet. Use only one of either the <code><servlet-class></code> tags or <code><jsp-file></code> tags in your servlet body.
<code><jsp-file></code>	Required (or use <code><servlet-class></code>)	The full path to a JSP file within the Web application, relative to the Web application root directory. Use only one of either the <code><servlet-class></code> tags or <code><jsp-file></code> tags in your servlet body.
<code><init-param></code>	Optional	Contains a name/value pair as an initialization attribute of the servlet. Use a separate set of <code><init-param></code> tags for each attribute.
<code><load-on-startup></code>	Optional	WebLogic Server initializes this servlet when WebLogic Server starts up. The optional content 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 during application startup.

Element	Required/ Optional	Description
<code><run-as></code>	Optional	Specifies the run-as identity to be used for the execution of the Web application. It contains an optional description and the name of a security role.
<code><security-role-ref></code>	Optional	Used to link a security role name defined by <code><security-role></code> 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

This is an element within the [“servlet” on page A-8](#).

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 an `icon` element.

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

init-param

This is an element within the [“servlet” on page A-8](#).

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

You can access these attributes 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
<code><param-name></code>	Required	Defines the name of this attribute.
<code><param-value></code>	Required	Defines a <code>String</code> value for this attribute.
<code><description></code>	Optional	Text description of the initialization attribute.

security-role-ref

This is an element within the “[servlet](#)” on page A-8.

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

servlet-mapping

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
<code><servlet-name></code>	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 <code><servlet></code> declaration tag.
<code><url-pattern></code>	Required	<p>Describes a pattern used to resolve URLs. The portion of the URL after the <code>http://host:port + WebAppName</code> is compared to the <code><url-pattern></code> by WebLogic Server. If the patterns match, the servlet mapped in this element will be called.</p> <p>Example patterns:</p> <pre>/soda/grape/* /foo/* /contents *.foo</pre> <p>The URL must follow the rules specified in the Servlet 2.3 Specification.</p> <p>For additional examples of servlet mapping, see “Servlet Mapping” on page 3-2.</p>

session-config

The `session-config` element defines the session attributes for this Web application.

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

Element	Required/ Optional	Description
<code><session-timeout></code>	Optional	<p>The number of minutes after which sessions in this Web application expire. The value set in this element overrides the value set in the <code>TimeoutSecs</code> attribute of the <code><session-descriptor></code> element in the WebLogic-specific deployment descriptor <code>weblogic.xml</code>, unless one of the special values listed here is entered.</p> <p>Default value: -2</p> <p>Maximum value: <code>Integer.MAX_VALUE ÷ 60</code></p> <p>Special values:</p> <ul style="list-style-type: none"> • -2 = Use the value set by <code>TimeoutSecs</code> in <code><session-descriptor></code> element of <code>weblogic.xml</code> • -1 = Sessions do not timeout. The value set in <code><session-descriptor></code> element of <code>weblogic.xml</code> is ignored. <p>For more information, see “session-descriptor” on page B-5.</p>

mime-mapping

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

welcome-file-list

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 [“Configuring Welcome Pages” on page 3-7](#) and ["How WebLogic Server Resolves HTTP Requests."](#)

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

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

error-page

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” on page 3-9](#) and ["How WebLogic Server Resolves HTTP Requests."](#)

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

Note: Define either an `<error-code>` or an `<exception-type>` but not both.

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

taglib

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
<code><taglib-location></code>	Required	Gives the file name of the tag library descriptor relative to the root of the Web application. It is a good idea to store the tag library descriptor file under the <code>WEB-INF</code> directory so it is not publicly available over an HTTP request.
<code><taglib-uri></code>	Required	Describes a URI, relative to the location of the <code>web.xml</code> document, identifying a Tag Library used in the Web application. If the URI matches the URI string used in the <code>taglib</code> directive on the JSP page, this <code>taglib</code> is used.

resource-env-ref

The `resource-env-ref` element contains a declaration of a Web application's reference to an administered object associated with a resource in the Web application's environment. It consists of an optional description, the resource environment reference name, and an indication of the resource environment reference type expected by the Web application code.

For example:

```
<resource-env-ref>
    <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

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

Element	Required/Optional	Description
<code><description></code>	Optional	Provides a description of the resource environment reference.
<code><resource-env-ref-name></code>	Required	Specifies the name of a resource environment reference; its value is the environment entry name used in the Web application code. The name is a JNDI name relative to the <code>java:comp/env</code> context and must be unique within a Web application.
<code><resource-env-ref-type></code>	Required	Specifies the type of a resource environment reference. It is the fully qualified name of a Java language class or interface.

resource-ref

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
<code><description></code>	Optional	A text description.
<code><res-ref-name></code>	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.
<code><res-type></code>	Required	The Java type of the resource that corresponds to the reference name. Use the full package name of the Java type.

Element	Required/Optional	Description
<res-auth>	Required	Used to control the resource sign on for security. If set to APPLICATION, 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” on page A-19 .
<res-sharing-scope>	Optional	Specifies whether connections obtained through the given resource manager connection factory reference can be shared. Valid values: <ul style="list-style-type: none"> • Shareable • Unshareable

security-constraint

The `security-constraint` element defines the access privileges to a collection of resources defined by the `<web-resource-collection>` element.

For detailed instructions and an example on configuring security in Web applications, see [Securing WebLogic Resources](#). Also, for more information on WebLogic Security, refer to [Programming WebLogic 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 to which this security constraint is applied.
<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” on page A-17 .
<user-data-constraint>	Optional	Defines how the client should communicate with the server. See also “user-data-constraint” on page A-18 .

web-resource-collection

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

This is an element within the [“security-constraint” on page A-16](#).

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

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

auth-constraint

This is an element within the [“security-constraint” on page A-16](#).

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
<code><description></code>	Optional	A text description of this security constraint.
<code><role-name></code>	Optional	Defines which security roles can access resources defined in this security-constraint. Security role names are mapped to principals using the <code>security-role-ref</code> . See “ security-role-ref ” on page A-10.

user-data-constraint

This is an element within the “[security-constraint](#)” on page A-16.

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
<code><description></code>	Optional	A text description.
<code><transport-guarantee></code>	Required	<p>Specifies that the communication between client and server. WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the <code>INTEGRAL</code> or <code>CONFIDENTIAL</code> transport guarantee.</p> <p>Range of values:</p> <ul style="list-style-type: none"> <code>NONE</code>—The application does not require any transport guarantees. <code>INTEGRAL</code>—The application requires that the data be sent between the client and server in such a way that it cannot be changed in transit. <code>CONFIDENTIAL</code>—The application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission.

login-config

Use the optional `login-config` element to configure 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
<code><auth-method></code>	Optional	Specifies the method used to authenticate the user. Possible values: BASIC—uses browser authentication. (This is the default value.) FORM—uses a user-written HTML form. CLIENT-CERT
<code><realm-name></code>	Optional	The name of the realm that is referenced to authenticate the user credentials. If omitted, the realm defined with the Auth Realm Name field on the Web application → Configuration → Other tab of the Administration Console is used by default. <i>For more information, see "Managing WebLogic Security".</i> Note: The <code><realm-name></code> element does not refer to system security realms within WebLogic Server. This element defines the realm name to use in HTTP Basic authorization. The system security realm is a collection of security information that is checked when certain operations are performed in the server. The servlet security realm is a different collection of security information that is checked when a page is accessed and basic authentication is used.
<code><form-login-config></code>	Optional	Use this element if you configure the <code><auth-method></code> to FORM. See “form-login-config” on page A-19 .

form-login-config

This is an element within the [“login-config” on page A-19](#).

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

Element	Required/ Optional	Description
<code><form-login-page></code>	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-based authentication that conforms to a specific naming convention.
<code><form-error-page></code>	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

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

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

env-entry

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 an `env-entry` element.

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

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

ejb-ref

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 an `ejb-ref` element.

Element	Required/ Optional	Description
<code><description></code>	Optional	A text description of the reference.
<code><ejb-ref-name></code>	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 <code>weblogic.xml</code> . For more information, see “ejb-reference-description” on page B-5 .
<code><ejb-ref-type></code>	Required	The expected Java class type of the referenced EJB.
<code><home></code>	Required	The fully qualified class name of the EJB home interface.
<code><remote></code>	Required	The fully qualified class name of the EJB remote interface.

Element	Required/ Optional	Description
<code><ejb-link></code>	Optional	The <code><ejb-name></code> of an EJB in an encompassing J2EE application package.
<code><run-as></code>	Optional	A security role whose security context is applied to the referenced EJB. Must be a security role defined with the <code><security-role></code> element.

ejb-local-ref

The `ejb-local-ref` element is used for the declaration of a reference to an enterprise bean's local home. The declaration consists of:

- An optional description
- The EJB reference name used in the code of the Web application that references the enterprise bean. The expected type of the referenced enterprise bean
- The expected local home and local interfaces of the referenced enterprise bean
- Optional `ejb-link` information, used to specify the referenced enterprise bean

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

Element	Required/ Optional	Description
<code><description></code>	Optional	A text description of the reference.
<code><ejb-ref-name></code>	Required	Contains the name of an EJB reference. The EJB reference is an entry in the Web application's environment and is relative to the <code>java:comp/env</code> context. The name must be unique within the Web application. It is recommended that name is prefixed with <code>ejb/</code> . For example: <code><ejb-ref-name>ejb/Payroll</ejb-ref-name></code>
<code><ejb-ref-type></code>	Required	The <code>ejb-ref-type</code> element contains the expected type of the referenced enterprise bean. The <code>ejb-ref-type</code> element must be one of the following: <code><ejb-ref-type>Entity</ejb-ref-type></code> <code><ejb-ref-type>Session</ejb-ref-type></code>

Element	Required/Optional	Description
<code><local-home></code>	Required	Contains the fully-qualified name of the enterprise bean's local home interface.
<code><local></code>	Required	Contains the fully-qualified name of the enterprise bean's local interface.
<code><ejb-link></code>	Optional	<p>The <code>ejb-link</code> element is used in the <code>ejb-ref</code> or <code>ejb-local-ref</code> elements to specify that an EJB reference is linked to an EJB.</p> <p>The name in the <code>ejb-link</code> element is composed of a path name. This path name specifies the <code>ejb-jar</code> containing the referenced EJB with the <code>ejb-name</code> of the target bean appended and separated from the path name by #.</p> <p>The path name is relative to the WAR file containing the Web application that is referencing the EJB. This allows multiple EJBs with the same <code>ejb-name</code> to be uniquely identified.</p> <p>Used in: <code>ejb-local-ref</code> and <code>ejb-ref</code> elements</p> <p>Examples:</p> <pre><ejb-link>EmployeeRecord</ejb-link></pre> <pre><ejb-link>../products/product.jar#ProductEJB</ejb-link></pre>

web.xml Deployment Descriptor Elements

weblogic.xml Deployment Descriptor Elements

The following sections describe the deployment descriptor elements that you define in the `weblogic.xml` file under the root element `<weblogic-web-app>`:

- “`auth-filter`” on page B-15
- “`charset-params`” on page B-17
- “`container-descriptor`” on page B-15
- “`context-root`” on page B-22
- “`description`” on page B-2
- “`destroy-as`” on page B-24
- “`init-as`” on page B-23
- “`jsp-descriptor`” on page B-11
- “`preprocessor`” on page B-20
- “`preprocessor-mapping`” on page B-21
- “`reference-descriptor`” on page B-4
- “`run-as-role-assignment`” on page B-3
- “`security-permission`” on page B-21
- “`security-role-assignment`” on page B-2

- “servlet-descriptor” on page B-22
- “session-descriptor” on page B-5
- “url-match-map” on page B-20
- “virtual-directory-mapping” on page B-19
- “weblogic-version” on page B-2
- “wl-dispatch-policy” on page B-22

The DOCTYPE header for the `weblogic.xml` file is as follows:

```
<!DOCTYPE weblogic-web-app PUBLIC
    "-//BEA Systems, Inc.//DTD Web Application 8.1//EN"
    "http://www.bea.com/servers/wls810/dtd/weblogic810-web-jar.dtd">
```

You can also access the Document Type Descriptor (DTD) for `weblogic.xml` at <http://www.bea.com/servers/wls810/dtd/weblogic810-web-jar.dtd>.

description

The `description` element is a text description of the Web application.

weblogic-version

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

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
<code><role-name></code>	Required	Specifies the name of a security role.
<code><principal-name></code>	Required	Specifies the name of a principal that is defined in the security realm. You can use multiple <code><principal-name></code> elements to map principals to a role. For more information on security realms, see Managing WebLogic Security .
<code><externally-defined></code>	Optional	Specifies that a particular security role is defined globally in a security realm; WebLogic Server uses this security role as the principal name, rather than looking it up in a global realm. When the security role and its principal-name mapping are defined elsewhere, this is used as an indicative placeholder.

Note: If you do not define a `security-role-assignment` element and its subelements, the Web application container implicitly maps the role name as a principal name and logs a warning. The EJB container does not deploy the module if mappings are not defined.

Consider the following usage scenarios for the role name is “role_xyz”

- If you map “role_xyz to user “joe” in `weblogic.xml`, `role_xyz` becomes a local role.
- If you specify `role_xyz` as an externally defined role, it becomes global (it refers to the role defined at the realm level).
- If you do not define a `security-role-assignment` element, `role_xyz` becomes a local role, and the Web application container creates an implicit mapping to it and logs a warning.

run-as-role-assignment

The `run-as-role-assignment` element maps a `run-as` role name (a subelement of the `servlet` element) in `web.xml` to a valid user name in the system. The value can be overridden for a given `servlet` by the `run-as-principal-name` element in the `servlet-descriptor`. If the `run-as-role-assignment` is absent for a given role name, the Web application container chooses the first `principal-name` defined in the `security-role-assignment`.

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

Element	Required/Optional	Description
<code><role-name></code>	Required	Specifies the name of a security role.
<code><run-as-principal-name></code>	Required	Specifies the name of a principal.

reference-descriptor

The `reference-descriptor` element maps a name used in the Web application to the JNDI name of a server resource. 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-env-description

The `resource-env-description` element maps a `resource-env-ref`, declared in the `ejb-jar.xml` deployment descriptor, to the JNDI name of the server resource it represents.

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

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

resource-description

The `resource-description` element is used to map the JNDI name of a server resource to an EJB resource reference in WebLogic Server.

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

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

ejb-reference-description

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

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

session-descriptor

The `session-descriptor` element contains the `session-param` element, which defines attributes 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-param

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

Element	Required/ Optional	Description
<code><param-name></code>	Required	Specifies the name of the parameter.
<code><param-value></code>	Required	Specifies the value of the parameter.

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

Attribute Name	Default Value	Value
<code>ConsoleMainAttribute</code>		If you enable Session Monitoring in the WebLogic Server Administration Console, set this attribute to the name of the session attribute you will use to identify each session that is monitored.
<code>CookieDomain</code>	Null	Specifies the domain for which the cookie is valid. For example, setting <code>CookieDomain</code> to <code>.mydomain.com</code> returns cookies to any server in the <code>*.mydomain.com</code> domain. The domain name must have at least two components. Setting a name to <code>*.com</code> or <code>*.net</code> is not valid. If not set, this attribute defaults to the server that issued the cookie. For more information, see <code>Cookie.setDomain()</code> in the Servlet specification from Sun Microsystems.
<code>CookieComment</code>	Weblogic Server Session Tracking Cookie	Specifies the comment that identifies the session tracking cookie in the cookie file. If not set, this attribute defaults to <code>WebLogic Session Tracking Cookie</code> . You may provide a more specific name for your application.

Attribute Name	Default Value	Value
CookieSecure	false	<p>Tells the browser to only send the cookie back over an HTTPS connection. This ensures that the cookie ID is secure and should only be used on Websites that use HTTPS. Session Cookies over HTTP no longer work if this feature is enabled.</p> <p>You should turn off the <code>URLRewritingEnabled</code> attribute if you intend to use this feature.</p>
CookieMaxAgeSecs	-1	<p>Sets the life span of the session cookie, in seconds, after which it expires on the client.</p> <p>If the value is 0, the cookie expires immediately.</p> <p>The maximum value is <code>Integer.MAX_VALUE</code>, where the cookie lasts forever.</p> <p>If set to <code>-1</code>, the cookie expires when the user exits the browser.</p> <p>For more information about cookies, see “Using Sessions and Session Persistence in Web Applications” on page 4-1.</p>
CookieName	JSESSIONID	<p>Defines the session cookie name. Defaults to <code>JSESSIONID</code> if not set. You may set this to a more specific name for your application.</p>
CookiePath	Null	<p>Specifies the path name to which the browser sends cookies.</p> <p>If not set, this attribute defaults to <code>/</code> (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.</p>
CookiesEnabled	true	<p>Use of session cookies is enabled by default and is recommended, but you can disable them by setting this property to <code>false</code>. You might turn this option off to test.</p>

Attribute Name	Default Value	Value
EncodeSessionIdInQueryParams	false	<p>By default, when you use the <code>HTTPServletResponse.encodeURL(URL)</code> method to encode a URL in the HTTP response, the session identifier is added to the URL as a path parameter after the <code>;</code> character in the URL. This behavior is defined by the Servlet 2.3 J2EE specification, implemented as of Version 6.1 of WebLogic Server.</p> <p>In Versions 6.0 and previous of WebLogic Server, however, the default behavior was to add the session identifier as a query parameter after the <code>?</code> character in the URL. To enable this old behavior, set this session parameter to <code>true</code>.</p> <p>Note: You typically use this parameter when WebLogic Server interacts with Web Servers that do not completely comply with the Servlet 2.3 specification.</p>
IDLength	52	<p>Sets the size of the session ID.</p> <p>The minimum value is 8 bytes and the maximum value is <code>Integer.MAX_VALUE</code>.</p> <p>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 attributes), which limits the amount of data that can be transmitted using URL re-writing. To allow more space for attributes, use this attribute to limit the size of the session ID that is randomly generated by WebLogic Server.</p> <p>You can also limit the length to a fixed 52 characters, and disallow special characters, by setting the <code>WAPEnabled</code> attribute. For more information, see URL Rewriting and Wireless Access Protocol in <i>Developing Web Applications for WebLogic Server</i>.</p>

Attribute Name	Default Value	Value
<code>InvalidationIntervalSecs</code>	60	<p>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 attribute to tune WebLogic Server for best performance on high traffic sites.</p> <p>The minimum value is every second (1). The maximum value is once a week (604,800 seconds). If not set, the attribute defaults to 60 seconds.</p>
<code>JDBCConnectionTimeoutSecs</code>	120	<p>Sets the time, in seconds, that WebLogic Server waits before timing out a JDBC connection, where x is the number of seconds between.</p>
<code>PersistentStoreDir</code>	<code>session_db</code>	<p>If you have set <code>PersistentStoreType</code> to <code>file</code>, this attribute sets the directory path where WebLogic Server will store the sessions. The directory path is an absolute path to the temp directory. The temp directory is specified by the <code>context-param</code> <code>javax.servlet.context.tmpdir</code>.</p> <p>Ensure that you have enough disk space to store the <i>number of valid sessions</i> multiplied by the <i>size of each session</i>. You can find the size of a session by looking at the files created in the <code>PersistentStoreDir</code>. Note that the size of each session can vary as the size of serialized session data changes.</p> <p>You can make file-persistent sessions clusterable by making this directory a shared directory among different servers.</p> <p>You must create this directory manually.</p>
<code>PersistentStorePool</code>	None	<p>Specifies the name of a JDBC connection pool to be used for persistence storage.</p>
<code>PersistentStoreTable</code>	<code>wl_servlet_sessions</code>	<p>Applies only when <code>PersistentStoreType</code> is set to <code>jdbc</code>. This is used when you choose a database table name other than the default.</p>

weblogic.xml Deployment Descriptor Elements

Attribute Name	Default Value	Value
PersistentStoreType	memory	<p>Sets the persistent store method to one of the following options:</p> <ul style="list-style-type: none">• <code>memory</code>—Disables persistent session storage.• <code>file</code>—Uses file-based persistence (See also <code>PersistentStoreDir</code>, above).• <code>jdbc</code>—Uses a database to store persistent sessions. (see also <code>PersistentStorePool</code>, above).• <code>replicated</code>—Same as <code>memory</code>, but session data is replicated across the clustered servers.• <code>cookie</code>—All session data is stored in a cookie in the user’s browser.• <code>replicated_if_clustered</code>—If the Web application is deployed on a clustered server, the in-effect <code>PersistentStoreType</code> will be replicated. Otherwise, <code>memory</code> is the default.
PersistentStoreCookieName	WLCOOKIE	<p>Sets the name of the cookie used for cookie-based persistence. The <code>WLCOOKIE</code> cookie carries the session state, which should not be shared between Web applications.</p> <p>For more information, see “Using Cookie-Based Session Persistence” on page 4-9.</p>

Attribute Name	Default Value	Value
TimeoutSecs	3600	<p>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.</p> <p>Minimum value is 1, default is 3600, and maximum value is integer MAX_VALUE.</p> <p>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.</p> <p>This attribute can be overridden by the <code>session-timeout</code> element (defined in minutes) in <code>web.xml</code>. For more information, see “session-config” on page A-11.</p>
TrackingEnabled	true	<p>Tells the Web application to keep track of the session between requests, in one of the following ways:</p> <p><code>SessionCookie</code></p> <p><code>URLEncoding</code></p> <p>If this is set to false, the session is not tracked, cookies coming in with the response are ignored, and the URL is not encoded.</p>
URLRewritingEnabled	true	<p>Enables URL rewriting, which encodes the session ID into the URL and provides session tracking if cookies are disabled in the browser.</p>

jsp-descriptor

The `jsp-descriptor` element defines attribute names and values for JSPs. You define the attributes as name/value pairs. The following example shows how to configure the `compileCommand` attribute. Enter all of the JSP configurations using the pattern demonstrated in this example:

```
<jsp-descriptor>
  <jsp-param>
    <param-name>
```

weblogic.xml Deployment Descriptor Elements

```
        compileCommand
    </param-name>
    <param-value>
        sj
    </param-value>
</jsp-param>
</jsp-descriptor>
```

The following table describes the element you can define within a `jsp-descriptor` element.

Element	Required/ Optional	Description
<code><jsp-param></code>	Required	Specifies parameters for servlet JSPs. Contains the following sub-elements: <ul style="list-style-type: none"><code><param-name></code> specifies a parameter name.<code><param-value></code> specifies a parameter value.

JSP Attribute Names and Values

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

Attribute Name	Default Value	Value
<code>compileCommand</code>	javac, or the Java compiler defined for a server under the configuration /tuning tab of the WebLogic Server Administration Console	Specifies the full path name 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: <pre><param-value> /jdk130/bin/javac.exe </param-value></pre> For faster performance, specify a different compiler, such as IBM Jikes or Symantec sj.
<code>compileFlags</code>	None	Passes one or more command-line flags to the compiler. Enclose multiple flags in quotes, separated by a space. For example: <pre><jsp-param> <param-name>compileFlags</param-name> <param-value>"-g -v"</param-value> </jsp-param></pre>
<code>compilerclass</code>	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.) If this attribute is set, the <code>compileCommand</code> attribute is ignored.
<code>debug</code>	None	When set to true this adds JSP line numbers to generated class files to aid debugging.
<code>encoding</code>	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 not set, this attribute defaults to the encoding for your platform. A JSP page directive (included in the JSP code) overrides this setting. For example: <pre><%@ page contentType="text/html; charset=custom-encoding"%></pre>

weblogic.xml Deployment Descriptor Elements

Attribute Name	Default Value	Value
compilerSupports Encoding	true	When set to true, the JSP compiler uses the encoding specified with the <code>contentType</code> attribute contained in the <code>page</code> directive on the JSP page, or, if a <code>contentType</code> is not specified, the encoding defined with the <code>encoding</code> attribute in the <code>jsp-descriptor</code> . When set to false, the JSP compiler uses the default encoding for the JVM when creating the intermediate <code>.java</code> file.
exactMapping	true	When true, upon the first request for a JSP the newly created <code>JspStub</code> is mapped to the exact request. If <code>exactMapping</code> is set to false, the Web application container generates non-exact url mapping for JSPs. <code>exactMapping</code> allows path info for JSP pages.
keepgenerated	false	Saves the Java files that are generated as an intermediary step in the JSP compilation process. Unless this attribute is set to true, the intermediate Java files are deleted after they are compiled.
noTryBlocks	false	If a JSP file has numerous or deeply nested custom JSP tags and you receive a <code>java.lang.VerifyError</code> 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. This default is preset for a development environment. If set to -1, page checking and recompiling is disabled. In a production environment where <code>c</code> changes to a JSP are rare, change the value of <code>pageCheckSeconds</code> to 60 or greater, according to your tuning requirements, or to -1 to disable page checking and recompiling.
precompile	false	When set to true, WebLogic Server automatically precompiles all modified JSPs when the Web application is deployed or re-deployed or when starting WebLogic Server.

Attribute Name	Default Value	Value
precompileContinue	false	When set to true, WebLogic Server continues precompiling all modified JSPs even if some of those JSPs fail during compilation. Only takes effect when precompile is set to true.
printNulls	null	When set to false, this parameter ensures that expressions with “null” results are printed as “ “.
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.
compiler	javac	Sets the JSP compiler for use with this instance of WebLogic Server.
superclass	weblogic.servlet.jsp.JspBase	Provides a means to override the default superclass for JSPs. The JSPs are compiled as servlet classes extending from this base class.

auth-filter

The `auth-filter` element specifies an authentication filter `HttpServlet` class.

container-descriptor

The `<container-descriptor>` element defines general attributes for Web applications.

check-auth-on-forward

Add the `<check-auth-on-forward/>` element when you want to require authentication of forwarded requests from a servlet or JSP. Omit the tag if you do not want to require re-authentication. For example:

```
<container-descriptor>
  <check-auth-on-forward/>
</container-descriptor>
```

Note that the default behavior has changed with the release of the Servlet 2.3 specification, which states that authentication is not required for forwarded requests.

redirect-with-absolute-url

The `<redirect-with-absolute-url>` element controls whether the `javax.servlet.http.HttpServletResponse.sendRedirect()` method redirects using a relative or absolute URL. Set this element to `false` if you are using a proxy HTTP server and do not want the URL converted to a non-relative link.

The default behavior is to convert the URL to a non-relative link.

user readable data used in a redirect.

index-directory-enabled

The `<index-directory-enabled>` element controls whether or not to automatically generate an HTML directory listing if no suitable index file is found.

The default value is `false` (does not generate a directory). Values are `true` or `false`.

index-directory-sort-by

The `<index-directory-sort-by>` element defines the order in which the directory listing generated by `weblogic.servlet.FileServlet` is sorted. Valid sort-by values are `NAME`, `LAST_MODIFIED`, and `SIZE`. The default sort-by value is `NAME`.

servlet-reload-check-secs

The `<servlet-reload-check-secs>` element defines whether a WebLogic Server will check to see if a servlet has been modified, and if it has been modified, reloads it. The `-1` value tells the server never to check the servlets, `0` tells the server to always check the servlets, and the default is to check each 1 second.

A value specified in the console will always take precedence over a manually specified value.

single-threaded-servlet-pool-size

The `<single-threaded-servlet-pool-size>` element defines the size of the pool used for `SingleThreadMode` instance pools. The default value is `5`.

session-monitoring-enabled

The `<session-monitoring-enabled>` element, if set to true, allows runtime MBeans to be created for sessions. When set to false, the default value, runtime MBeans are not created. A value specified in the console takes precedence over a value set manually.

save-sessions-enabled

The `<save-sessions-enabled>` element controls whether session data is cleaned up during redeploy or undeploy. It affects memory and replicated sessions. Setting the value to true means session data is saved. Setting to false means session data will be destroyed when the Web application is redeployed or undeployed. The default is false.

prefer-web-inf-classes

The `<prefer-web-inf-classes>` element, if set to true, will cause classes located in the WEB-INF directory of a Web application to be loaded in preference to classes loaded in the application or system classloader. The default value is false. A value specified in the console will take precedence over a value set manually.

default-mime-type

The `<default-mime-type>` element default value is null. This element allows the user to specify the default mime type for a content-type for which the extension is not mapped.

retain-original-url

Set the `<retain-original-url>` element to true to retain the HTTP in the original URL you are requesting prior to being forwarded to the authentication URL.

Once you login successfully using the authentication URL, you are then taken back to the exact URL that you had originally requested.

charset-params

The `<charset-params>` element is used to define code set behavior for non-unicode operations. For example:

```
<charset-params>  
  <input-charset>
```

```

    <resource-path>/*</resource-path>
    <java-charset-name>UTF-8</java-charset-name>
  </input-charset>
</charset-params>

```

input-charset

Use the `<input-charset>` element to define which character set is used to read GET and POST data. For example:

```

<input-charset>
  <resource-path>/foo</resource-path>
  <java-charset-name>SJIS</java-charset-name>
</input-charset>

```

For more information, see [“Loading Servlets, Context Listeners, and Filters” on page 3-18](#).

The following table describes the elements you can define within a `<input-charset>` element.

Element	Required/ Optional	Description
<code><resource-path></code>	Required	A path which, if included in the URL of a request, signals WebLogic Server to use the Java character set specified by <code><java-charset-name></code> .
<code><java-charset-name></code>	Required	Specifies the Java characters set to use.

charset-mapping

Use the `<charset-mapping>` element to map an IANA character set name to a Java character set name. For example:

```

<charset-mapping>
  <iana-charset-name>Shift-JIS</iana-charset-name>
  <java-charset-name>SJIS</java-charset-name>
</charset-mapping>

```

For more information, see [“Mapping IANA Character Sets to Java Character Sets” on page 3-20](#).

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

Element	Required/ Optional	Description
<code><iana-charset-name></code>	Required	Specifies the IANA character set name that is to be mapped to the Java character set specified by the <code><java-charset-name></code> element.
<code><java-charset-name></code>	Required	Specifies the Java characters set to use.

virtual-directory-mapping

Use the `virtual-directory-mapping` element to specify document roots other than the default document root of the Web application for certain kinds of requests, such as image requests. All images for a set of Web applications can be stored in a single location, and need not be copied to the document root of each Web application that uses them. For an incoming request, if a virtual directory has been specified servlet container will search for the requested resource first in the virtual directory and then in the Web application's original document root. This defines the precedence if the same document exists in both places.

Example:

```
<virtual-directory-mapping>
    <local-path>c:/usr/gifs</local-path>
    <url-pattern>/images/*</url-pattern>
    <url-pattern>*.jpg</url-pattern>
</virtual-directory-mapping>
<virtual-directory-mapping>
    <local-path>c:/usr/common_jsps.jar</local-path>
    <url-pattern>*.jsp</url-pattern>
</virtual-directory-mapping>
```

The following table describes the elements you can define within the `virtual-directory-mapping` element.

Element	Required/ Optional	Description
<code><local-path></code>	Required	Specifies a physical location on the disk.
<code><url-pattern></code>	Required	Contains the URL pattern of the mapping. Must follow the rules specified in Section 11.2 of the Servlet API Specification.

The WebLogic Server implementation of virtual directory mapping requires that you have a directory that matches the `url-pattern` of the mapping. The image example requires that you create a directory named `images` at `c:/usr/gifs/images`. This allows the servlet container to find images for multiple Web applications in the `images` directory.

url-match-map

Use this element to specify a class for URL pattern matching. The WebLogic Server default URL match mapping class is `weblogic.servlet.utils.URLMatchMap`, which is based on J2EE standards. Another implementation included in WebLogic Server is `SimpleApacheURLMatchMap`, which you can plug in using the `url-match-map` element.

Rule for `SimpleApacheURLMatchMap`:

If you map `*.jws` to `JWSServlet` then

`http://foo.com/bar.jws/baz` will be resolved to `JWSServlet` with `pathInfo = baz`.

Configure the `URLMatchMap` to be used in `weblogic.xml` as in the following example:

```
<url-match-map>
  weblogic.servlet.utils.SimpleApacheURLMatchMap
</url-match-map>
```

preprocessor

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

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

Element	Required/ Optional	Description
<preprocessor-name>	Required	Contains the canonical name of the preprocessor.
<preprocessor-class>	Required	Contains the fully qualified class name of the preprocessor.

preprocessor-mapping

The `preprocessor-mapping` element defines a mapping between a preprocessor and a URL pattern.

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

Element	Required/ Optional	Description
<preprocessor-name>	Required	
<url-pattern>	Required	

security-permission

The `security-permission` element specifies a single security permission based on the Security policy file syntax. Refer to the following URL for Sun's implementation of the security permission specification:

<http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html#FileSyntax>

Disregard the optional `codebase` and `signedBy` clauses.

For example:

```
<security-permission-spec>
    grant { permission java.net.SocketPermission "*", "resolve" };
</security-permission-spec>
```

where:

`permission java.net.SocketPermission` is the permission class name.

`"*"` represents the target name.

`resolve` indicates the action.

context-root

The `context-root` element defines the context root of this stand-alone Web application. If the Web application is part of an EAR, not stand-alone, specify the context root in the EAR's `application.xml` file. A `context-root` setting in `application.xml` takes precedence over `context-root` setting in `weblogic.xml`.

Note that this `weblogic.xml` element only acts on deployments using the two-phase deployment model. See ["Two-Phase Deployment"](#) in *Deploying WebLogic Server Applications*.

The order of precedence for context root determination for a Web application is as follows:

1. Check `application.xml` for context root; if found, use as Web application's context root.
2. If context root is not set in `application.xml`, and the Web application is being deployed as part of an EAR, check whether context root is defined in `weblogic.xml`. If found, use as Web application's context root. If the Web application is deployed standalone, `application.xml` does not come into play and the determination for context-root starts at `weblogic.xml` and defaults to URI if it is not defined there.
3. If context root is not defined in `weblogic.xml` or `application.xml`, then infer the context path from the URI, giving it the name of the value defined in the URI minus the WAR suffix. For instance, a URI `MyWebApp.war` would be named `MyWebApp`.

wl-dispatch-policy

Use the `wl-dispatch-policy` element to assign the Web application to a configured execute queue by identifying the execute queue name.

servlet-descriptor

Use the `servlet-descriptor` element to aggregate the servlet-specific elements.

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

Element	Required/Optional	Description
<code><servlet-name></code>	Required	Specifies the servlet name as defined in the <code>servlet</code> element of the <code>web.xml</code> deployment descriptor file.
<code><run-as-principal-name></code>	Optional	Contains the name of a principal against the <code>run-as-role-name</code> defined in the <code>web.xml</code> deployment descriptor.
<code><init-as-principal-name></code>	Optional	Equivalent to <code>run-as-principal-name</code> for the <code>init</code> method for servlets. The identity specified here should be a valid user name in the system. If <code>init-as-principal-name</code> is not specified, the container uses the <code>run-as-principal-name</code> element.
<code><destroy-as-principal-name></code>	Optional	Equivalent to <code>run-as-principal-name</code> for the <code>destroy</code> method for servlets. The identity specified here should be a valid user name in the system. If <code>destroy-as-principal-name</code> is not specified, the container uses the <code>run-as-principal-name</code> element.
<code><dispatch-policy></code>	Optional	<i>This is a deprecated element.</i> Used to assign a given servlet to a configured <code>execute-queue</code> by identifying the <code>execute-queue</code> name. This setting overrides the Web application-level dispatch policy defined by <code>wl-dispatch-policy</code> .

init-as

This is an equivalent of `<run-as>` for `init` method for servlets.

For example:

```
<init-as>
  <servlet-name>FooServlet</servlet-name>
  <principal-name>joe</principal-name>
</init-as>
```

destroy-as

This is an equivalent of `<run-as>` for destroy method for servlets.

For example:

```
<destroy-as>  
  <servlet-name>BarServlet</servlet-name>  
  <principal-name>bob</principal-name>  
</destroy-as>
```

Index

A

- application events 5-2
- application event listeners 5-2

C

- CGI 3-9
- chaining filters 6-5
- Configuration
 - JSP tag libraries 3-7
 - servlets 3-1
- cookies 4-2
 - URL rewriting 4-10
- customer support contact information xv

D

- default servlet 3-8
- deployment
 - options, for resource adapters 2-5
 - overview 2-2
- deployment descriptor
 - re-deployment 2-8
- deployment descriptors
 - basic conventions for manually editing 2-3
 - DOCTYPE header information 2-4
- documentation, where to find it xiv
- doFilter() 6-3

E

- error pages 3-9
- event listener
 - declaration 5-4

- event listeners
 - configuring 5-4
- events
 - declaration 5-4
- exploded directory format
 - re-deployment 2-6

F

- filter class 6-2
- filter mapping 6-4
 - to a servlet 6-5
 - URL pattern 6-4
- filters
 - and Web Applications 6-2
 - chaining 6-5
 - configuring 6-3
 - declaration 6-3
 - mappings 6-4
 - overview 6-1
 - uses 6-2
 - writing a filter class 6-2

H

- HTTP session events 5-3
- HTTP sessions 4-1
 - and redeployment 2-8

I

- init params 3-5
- in-memory replication 4-4

J

JSP

- modifying 2-8
- refreshing 2-8
- tag libraries 3-7

L

listener

- writing a listener class 5-5

listener class 5-5

listeners 5-2

- configuring 5-4
- HTTP session events 5-3
- servlet context events 5-2

M

manually editing XML deployment files 2-3

mapping

- filters 6-4

modifying components 2-8

modifying JSP 2-8

P

persistence for sessions 4-4

printing product documentation xiv

R

REDEPLOY file 2-7

re-deployment 2-5

- and HTTP sessions 2-8
- exploded directory format 2-6
- of Java classes 2-8
- using REDEPLOY file 2-7
- when using auto-deployment 2-5

refreshing

- JSP 2-8

resource adapters

- deployment options 2-5

response 6-1

S

servlet

- configuration 3-1
- default servlet 3-8
- initialization parameters 3-5
- mapping 3-2
- url-pattern 3-2

servlet context events 5-2

session persistence

- file-based 4-5
- JDBC (database) 4-6
- single server 4-5

Session Timeout 4-2

sessions 4-1

- cookies 4-2
- persistence 4-4
- Session Timeout attribute 4-2
- setting up 4-1
- URL rewriting 4-10
- URL rewriting and WAP 4-11

support

- technical xv

U

URL rewriting 4-10

W

WAP 4-11

Web Application

- configuring external resources 3-13
- default servlet 3-8
- error page 3-9
- URI 1-8

WEB-INF directory 1-4

weblogic-ra.xml file

- manually editing XML deployment files 2-3

welcome pages 3-7

X

XML deployment files, manually editing 2-3