BEA WebLogic Server™

Developing Web Applications for WebLogic Server

Release 8.1
Document Revised: December 3, 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Developing Web Applications for WebLogic Server

Part Number	Document Revised	Software Version
N/A	December 3, 2002	BEA WebLogic Server Version 8.1

Contents

About This Document

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

1. Web Applications Basics

How to Use This Book 1-1
Overview of Web Applications 1-2
    Servlets 1-3
    Java Server Pages 1-3
    Web Application Directory Structure 1-3
Main Steps to Create a Web Application 1-4
Directory Structure 1-6
URLs and Web Applications 1-7
Web Application Developer Tools 1-7
    WebLogic Builder 1-8
    Ant Tasks to Create Skeleton Deployment Descriptors 1-8
    BEA XML Editor 1-9
    appc and jspc Compilers 1-9
        appc Compiler 1-9
        jspc Compiler 1-11

2. Deploying Web Applications

Redeploying a Web Application Using Auto-Deployment 2-2

Redeploying a Web Application in a WAR Archive 2-2
Redeploying a Web Application in Exploded Directory Format 2-2
Touching the REDEPLOY File 2-2
Redeploying with the Administration Console 2-3
Hot-Deployment 2-3
Requirements for Redeploying a Web Application in Production Mode 2-4
Refreshing Static Components (JSP Files, HTML Files, Image Files, Etc.) 2-5
Deploying Web Applications as Part of an Enterprise Application 2-6
Deploying a Default Web Application 2-7

3. Configuring Web Application Components

Configuring Servlets 3-2
Servlet Mapping 3-2
Servlet Initialization Parameters 3-4
Configuring Java Server Pages (JSPs) 3-5
Registering JSPs as a Servlet 3-6
Configuring JSP Tag Libraries 3-6
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
Serving Resources from the CLASSPATH with the ClasspathServlet 3-12
Configuring Resources in a Web Application 3-12
Configuring External Resources 3-13
Configuring Application-Scoped Resources 3-14
Referencing EJBs in a Web Application 3-15
Referencing External EJBs 3-15
More about the ejb-ref* Elements 3-16
Referencing Application-Scoped EJBs 3-17
Determining the Encoding of an HTTP Request 3-20
Mapping IANA Character Sets to Java Character Sets 3-21

4. Using Sessions and Session Persistence in Web Applications

Overview of HTTP Sessions 4-1
Setting Up Session Management 4-2
HTTP Session Properties 4-2
Session Timeout 4-2
Configuring Session Cookies 4-3
Using Cookies That Outlive a Session 4-3
Logging Out and Ending a Session 4-4
Configuring Session Persistence 4-4
Common Properties of Session Attributes 4-5
Using Memory-based, Single-server, Non-replicated Persistent Storage .. 4-6
Using File-based Persistent Storage 4-6
Using a Database for Persistent Storage (JDBC persistence) 4-6
Using Cookie-Based Session Persistence 4-8
Using URL Rewriting 4-9
Coding Guidelines for URL Rewriting 4-9
URL Rewriting and Wireless Access Protocol (WAP) 4-10

5. Application Events and Listeners

Overview of Application Events and Listeners 5-1
Servlet Context Events 5-2
HTTP Session Events 5-3
Configuring an Event Listener 5-3
Writing a Listener Class 5-4
Templates for Listener Classes 5-5
Servlet Context Listener Example 5-5
HTTP Session Attribute Listener Example 5-6
Additional Resources 5-6

6. Configuring Security in Web Applications

Overview of Configuring Security in Web Applications 6-1
Setting Up Authentication for Web Applications 6-2
Multiple Web Applications, Cookies, and Authentication 6-4
Restricting Access to Resources in a Web Application 6-5
Using Users and Roles Programmatically in Servlets 6-6

7. Filters

Overview of Filters 7-1
How Filters Work 7-2
Uses for Filters 7-2
Configuring Filters 7-3
Configuring a Filter 7-3
Configuring a Chain of Filters 7-5
Writing a Filter 7-5
Example of a Filter Class 7-7
Filtering the Servlet Response Object 7-8
Additional Resources 7-8

A. web.xml Deployment Descriptor Elements

icon A-2
display-name A-3
description A-3
distributable A-3
context-param A-4
filter A-4
filter-mapping A-5
listener A-6
servlet A-6
icon A-8
init-param A-8
security-role-ref A-9
servlet-mapping A-10
session-config A-11
mime-mapping A-11
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-20
security-role A-20
env-entry A-21
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
reference-descriptor.....B-3
resource-description B-4
ejb-reference-description.....B-4
session-descriptor B-4
Session Parameter Names and Values B-5
jsp-descriptor.....B-9
JSP Parameter Names and Values.....B-9
auth-filter B-11
container-descriptor.....B-11
check-auth-on-forward.....B-12
redirect-content-type B-12
redirect-content.....B-12
redirect-with-absolute-url.....B-12
charset-params.....B-13
input-charset.....B-13
charset-mapping B-13
virtual-directory-mapping.....B-14
url-match-map B-15
preprocessor.....B-16
preprocessor-mapping B-16
security-permission.....B-17
context-root.....B-17

wl-dispatch-policy B-18
init-as B-19
destroy-as B-19
index-directory B-19

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 6, "Configuring Security in Web Applications," describes how to configure authentication and authorization in a Web Application.
- Chapter 5, "Application Events and Listeners," describes how to use J2EE event listeners in a Web Application.
- Chapter 7, "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.
italics	Emphasis and book titles.

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

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName`*`;`

Convention	Usage
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER

1 Web Applications Basics

The following sections describe how to configure and deploy Web Applications:

- "How to Use This Book" on page 1-1
- "Overview of Web Applications" on page 1-2
- "Main Steps to Create a Web Application" on page 1-4
- "Directory Structure" on page 1-6
- "URLs and Web Applications" on page 1-7
- "Web Application Developer Tools" on page 1-7

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 parameters.

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:

- At least 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 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.

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 JSP compiler, `weblogic.jspc`, translates JSPs into servlets. WebLogic Server automatically compiles JSPs if the servlet class file is not present or is older than the JSP source file.

You can also precompile JSPs and package the servlet class in 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. Deploying a Web Application in exploded directory format is recommended primarily for use while developing your application. Deploying a Web Application as a WAR file is recommended primarily for production environments.

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.

Clients can generally browse any location in a Web application with the exception of the `WEB-INF` 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.

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

Main Steps to Create a Web Application

The following steps summarize the procedure for creating a Web Application. You may want to use developer tools included with WebLogic Server for creating and configuring Web Applications. For more information, see "Web Application Developer Tools" on page 1-7.

To create a Web Application:

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

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

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

   For detailed information about creating servlets, see *Programming WebLogic HTTP Servlets* and *Programming WebLogic JSP*.

3. Compile the servlets into class files.

   For detailed information about compiling java code, refer to the section "Developing WebLogic Server Applications" in *Developing WebLogic Server Applications*.

For instructions on using the WebLogic Server appc compiler, see "appc and jspc Compilers" on page 1-9

4. Arrange the resources (servlets, JSPs, static files, and deployment descriptors) in the prescribed directory format. See "Directory Structure" on page 1-6.

5. Create the Web Application deployment descriptor (`web.xml`) and place the descriptor in the `WEB-INF` directory of the Web Application. In this step you register servlets, define servlet initialization parameters, register JSP tag libraries, define security constraints, and define other Web Application parameters.

   You can edit Web Application deployment descriptors manually or using WebLogic Builder. For more information, see *WebLogic Builder Online Help*.

   For detailed information on the elements in the `web.xml` descriptor, see Appendix A, "web.xml Deployment Descriptor Elements."

6. Create the WebLogic-specific deployment descriptor (`weblogic.xml`) and place the descriptor in the `WEB-INF` directory of the Web Application. In this step you define how WebLogic Server will define JSP properties, JNDI mappings, security role mappings, and HTTP session parameters.

   You can edit Web Application deployment descriptors manually or using WebLogic Builder. For more information, see *WebLogic Builder Online Help*.

   For detailed information on the elements the `weblogic.xml` descriptor, see Appendix B, "weblogic.xml Deployment Descriptor Elements."

7. Package the Web Application files into a WAR file. (During development, you may find it more convenient to update individual components of your Web Application in exploded directory format.)

   For detailed information about packaging Web Applications, see "WebLogic Server Application Packaging" in *Developing WebLogic Server Applications*.

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

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

   While you are testing the Web application, you might need to edit the Web application deployment descriptors. You can do this manually or use WebLogic Builder. For more information, see *WebLogic Builder Online Help*.

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

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

Directory Structure

Develop your Web Application within a specified directory structure so that it can be archived and deployed on WebLogic Server or another J2EE-compliant server. All servlets, classes, static files, and other resources belonging to a Web Application are organized under a directory hierarchy. The root directory of this hierarchy defines the document root of your Web Application. All files under this root directory can be served to the client, except for files under the special directory `WEB-INF`, located under the root directory.

Place private files in the `WEB-INF` directory, under the root directory. All files under `WEB-INF` are private, and are not served to a client.

`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`.

  `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.

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*.

BEA XML Editor

You can use the BEA XML Editor to create and edit XML files. You can also validate XML code according to a specified DTD or XML Schema. It can be used on Windows or Solaris machines and is downloadable from BEA Dev2Dev Online at: `http://dev2dev.bea.com/resourcelibrary/utilitiestools/xml.jsp?highlight=utilitiestools`.

appc and jspc Compilers

This section discusses the appc and jspc compilers. The appc compiler allows compilation and validation of J2EE EAR files, EJB JAR files, and Web Application WAR files, whereas the jspc compiler only allows compilation of JSPs.

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>
```

appc Options

The following are the available appc options:

Option	Description
`-print`	Prints the standard usage message.
`-version`	Prints `jspc` version information.
`-output <file>`	Specifies an alternate output archive or directory. If not set, the output is placed in the source archive or directory.
`-forceGeneration`	Forces generation of EJB and JSP classes. Without this flag, the classes may not be regenerated (if determined to be unnecessary).
`-lineNumbers`	Adds JSP line numbers to generated class files to aid in debugging.
`-basicClientJar`	Does not include deployment descriptors in client JARs generated for EJBs.
`-idl`	Generates IDL (interface definition language) for EJB remote interfaces.
`-idlOverwrite`	Always overwrites existing IDL files.
`-idlVerbose`	Displays verbose information for IDL generation.
`-idlNoValueTypes`	Does not generate valuetypes and the methods and attributes that contain them.
`-idlNoAbstractInterfaces`	Does not generate abstract interfaces and methods and attributes that contain them.
`-idlFactories`	Generates factory methods for valuetypes.
`-idlVisibroker`	Generates IDL somewhat compatible with Visibroker 4.5 C++.
`-idlOrbix`	Generates IDL somewhat compatible with Orbix 2000 2.0 C++.
`-idlDirectory <dir>`	Specifies the directory where IDL files will be created (default : target directory or JAR)

`-idlMethodSignatures <>`	Specifies the method signatures used to trigger IDL code generation.
`-iiop`	Generates CORBA stubs for EJBs.
`-iiopDirectory <dir>`	Specifies the directory where IIOP stub files will be written (default : target directory or JAR)
`-keepgenerated`	Keeps the generated .java files.
`-compiler <javac>`	Selects the Java compiler to use.
`-g`	Compiles debugging information into a class file.
`-O`	Compiles with optimization on.
`-nowarn`	Compiles without warnings.
`-verbose`	Compiles with verbose output.
`-deprecation`	Warns about deprecated calls.
`-normi`	Passes flags through to Symantec's sj.
`-J<option>`	Passes flags through to Java runtime.
`-classpath <path>`	Selects the classpath to use during compilation.
`-advanced`	Prints advanced usage options.

appc Ant Task

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

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

jspc Compiler

The jspc compiler is a Java Servlet Page (JSP)-specific compiler. It offers more options for compiling JSPs than appc, and provides more control over how the JSPs are compiled.

jspc Options

The following are the available jspc options:

Option	Description
`-print`	Prints the standard usage message.
`-version`	Prints `jspc` version information.
`-webapp <dir>`	Directory to be considered as the document root for resolving relative files.
`-verboseJspc`	Indicates whether jspc runs in verbose mode. The default is false.
`-keepgenerated`	Keeps the generated .java files.
`-compiler <javac>`	Indicates the Java compiler to use.
`-compilerclass <null>`	Loads the compiler as a class instead of an executable.
`-classpath <path>`	Classpath to use during compilation.
`-d <dir>`	Target (top-level) directory for compilation.
`-advanced`	Prints advanced usage options.

jspc Syntax

Use the following syntax to run jspc:

```
prompt> java weblogic.jspc [options] <jsp files>...
```

jspc Usage Scenario

To compile all of the JSPs in a particular directory, use the following syntax:

```
java weblogic.jspc -webapp . -verboseJspc -d .\WEB-INF\classes
```

where:

- `'.'` is the current directory.

- `-d` is the target directory for all the compiled JSPs.
- `-verboseJspc` is the verbose flag.

To compile only a few select JSP files, use the following syntax:

```
java weblogic.jspc -webapp . -verboseJspc foo.jsp abc.jsp
```

CHAPTER

2 Deploying Web Applications

WebLogic Server application deployment is covered in detail in *Deploying WebLogic Server Applications*. This section explains only deployment procedures that are specific to Web 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 create a Web Application, see "Main Steps to Create a Web Application" on page 1-4.

The following sections provide Web Application-specific information:

- Redeploying a Web Application Using Auto-Deployment
- Requirements for Redeploying a Web Application in Production Mode
- Refreshing Static Components (JSP Files, HTML Files, Image Files, Etc.)
- Deploying Web Applications as Part of an Enterprise Application

Redeploying a Web Application Using Auto-Deployment

When you modify a component of a Web Application (such as a JSP, HTML page, or Java class) that is deployed in the `applications` directory and you are using auto-deployment, the Web Application must be re-deployed in order for the changes to become effective. The procedure is different for Web Applications deployed as WARs and Web Applications deployed in exploded directory format.

Redeploying a Web Application in a WAR Archive

Modifying the archive file automatically triggers re-deployment of the Web Application. If an auto-deployed Web Application is targeted to any Managed Servers, the Web Application is also re-deployed on the Managed Servers.

Redeploying a Web Application in Exploded Directory Format

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

Touching the REDEPLOY File

To re-deploy a Web Application by modifying 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. Modify the `REDEPLOY` file by opening it, modifying the contents (adding a space character is the easiest way to do this), and then saving the file. Alternately, on UNIX machines, you can use the `touch` command on the `REDEPLOY` file. For example:

   ```
   touch
   user_domains/mydomain/applications/DefaultWebApp/WEB-INF/REDEPL
   OY
   ```

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

Redeploying with the Administration Console

To redeploy a Web Application using the Administration Console:

1. Expand the Deployments node in the left pane.
2. Select the Web Application node.
3. Select the Web Application you want to redeploy.
4. Click the Undeploy button in the application's table in the right-hand pane.
5. Click the Deployed button in the application's table in the right-hand pane.

Hot-Deployment

Redeploy files in the WEB-INF/classes directory in the following way. If a class is deployed in WEB-INF/classes, then simply copying a new version of the file with a later time stamp will cause the Web Application to reload everything in the WEB-INF/classes folder with a new classloader.

The frequency in which WLS will look at the filesystem is governed through the console. In the Deployments-->Web Applications tab, select your Web Application. Go to the Configuration tab, and Files subtab, and enter a value in seconds for the Reload Period.

Requirements for Redeploying a Web Application in Production Mode

To redeploy a Web Application with Production Mode enabled, you must start WebLogic Server with the `-DProductionModeEnabled=true` flag. When you modify a component (for instance, a servlet, JSP, or HTML page) of a Web Application on the Administration Server, 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.
- 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 Web Application.

Refreshing Static Components (JSP Files, HTML Files, Image Files, Etc.)

While the `weblogic.Deployer` utility can refresh static files in your deployed applications, the command-line syntax for invoking earlier refresh tools remains viable. If you have scripts that invoke the `WebAppComponentRefreshTool` or `weblogic.jspRefresh`, they will now invoke the refresh capability of the `weblogic.Deployer` utility.

For information about the `weblogic.Deployer` utility, see "Deployment Tools Reference" in *Deploying WebLogic Server Applications*.

Use `jspRefresh` to refresh deployed static files such as:

- JSPs
- HTML files
- Image files such as gif and jpg
- Text files

You cannot use this utility to refresh Java class files.

To use `jspRefresh`, *you must deploy the Web Application in exploded directory format*. The utility does not work for components archived in WAR files.

To refresh a static file:

1. Set up your development environment so that WebLogic Server classes are in your system `CLASSPATH` and the JDK is available. You can use the `setEnv` script located in the `config/`*`mydomain`* directory to set your environment.
2. Enter the following command:

   ```
   % java weblogic.deploy -url adminServerURL -username
   AdminUserName -jspRefreshFiles fileList
   -jspRefreshComponentName component refresh password application
   ```

   Where:

   - *`url`* is the URL of your WebLogic Administration Server.

- *AdminUserName* is the username for system administration.
- *fileList* is a comma-separated list of files to be refreshed. Wildcard characters (`*.jsp`, for example) are not supported.
- *component* is the name of the Web Application being refreshed.
- *password* is your system administration password.
- *application* is the name of an Enterprise Application that contains the Web Application being refreshed. If your Web Application is not part of an Enterprise Application, enter the name of the Web Application.

For example, the following command refreshes the files `HelloWorld.jsp` and `ball.gif` in the `myWebApp` Web Application:

```
java weblogic.deploy -url t3://localhost:7001
 -username myUsername -jspRefreshFiles HelloWorld.jsp,ball.gif
 -jspRefreshComponentName myWebApp refresh myPassword myWebApp
```

Note: Even though the syntax of the command says `-jspRefreshFiles` and `-jspRefreshComponentName`, you can refresh any static file using this command, not just JSP files.

Deploying Web Applications as Part of an Enterprise Application

You can deploy a Web Application as part of an Enterprise Application. An Enterprise Application is a J2EE deployment unit that bundles together Web Applications, EJBs, and Resource Adaptors into a single deployable unit. (For more information on Enterprise Applications, see Packaging Components and Applications at `http://e-docs.bea.com/wls/docs81b/programming/packaging.html`.) If you deploy a Web Application as part of an Enterprise Application, you can specify a string that is used in place of the actual name of the Web Application when WebLogic Server resolves a request for the Web Application. You specify the new name with the `<context-root>` element in the `application.xml` deployment descriptor for the Enterprise Application. For more information, see application.xml Deployment Descriptor Elements at `http://e-docs.bea.com/wls/docs81b/programming/app_xml.html`.

For example, for a Web Application called `oranges`, you would typically request a resource from the `oranges` Web Application with a URL such as:

```
http://host:port/oranges/catalog.jsp.
```

If the `oranges` Web Application is packaged in an Enterprise Application, you specify a value for the `<context-root>` as shown in the following example:

```
<module>
   <web>
      <web-uri>oranges.war</web-uri>
      <context-root>fruit</context-root>
   </web>
</module>
```

You then use the following URL to access the same resource from the `oranges` Web Application:

```
http://host:port/fruit/catalog.jsp
```

Note: You cannot deploy the same Web Application under more than one name in one Enterprise Application. You can, however, deploy the same Web Application under more than one name if each Web Application is packaged in a different Enterprise Application.

Deploying a Default Web Application

The default Web Application is presented to clients who do not specify a URI (or specify "/" as the URI). To configure a Web Application as a default Web Application, set the value of the `context-root` element to “/” in its deployment descriptor.

CHAPTER

3 Configuring Web Application Components

The following sections describe how to configure Web Application components:

- "Configuring Servlets" on page 3-2
- "Configuring Java Server Pages (JSPs)" on page 3-5
- "Configuring JSP Tag Libraries" on page 3-6
- "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
- "Referencing EJBs in a Web Application" on page 3-15
- "Determining the Encoding of an HTTP Request" on page 3-20
- "Mapping IANA Character Sets to Java Character Sets" on page 3-21

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 JSPs.) This element also contains definitions for initialization parameters 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.jcp.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`	The default servlet, if configured, or an HTTP 404 File Not Found error message. If the mapping for the `list` servlet had been `/seedlist*`, the `list` servlet would be invoked.
`http://host:port/mywebapp/seedlist/pear.abc`	`kiwi` If the mapping for the `list` servlet had been `/seedlist*`, the `list` servlet would be invoked.
`http://host:port/mywebapp/seeds`	`garden`
`http://host:port/mywebapp/seeds/index.html`	`garden`
`http://host:port/mywebapp/index.abc`	`kiwi`

Servlet Initialization Parameters

You define initialization parameters 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 Parameters 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 parameters in the `<jsp-descriptor>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. These parameters 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 parameters, see "JSP Parameter Names and Values" on page B-9.

Registering JSPs 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 on 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 implicity register JSPs, as illustrated in the provided 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 parameters, 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>
   <tablig-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 parameter-based flow control, and facilitate iterations over sets of objects. For more information, see:

- Using Custom WebLogic JSP Tags.
- Using WebLogic JSP Form Validation Tags.

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`

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 3-2.
2. Map your default servlet with a url-pattern of "`/`". This causes your default servlet to respond to all types of files except for those with extensions of `*.htm` or `*.html`, which are internally mapped to the FileServlet.

   If you also want your default servlet to respond to files ending in `*.htm` or `*.html`, then you must map those extensions to your default servlet, in addition to mapping "`/`". For instructions on mapping servlets, see "Configuring Servlets" on page 3-2.
3. 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`.

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

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 (see "Sample Web Application Deployment Descriptor Entries for Registering the CGIServlet" on page 3-10). For more information, see "Configuring Servlets" on page 3-2.

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 parameters 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 ";" (Windows) or a ":" (Unix). If you do not specify `cgiDir`, the directory defaults to a directory named `cgi-bin` under the Web Application root.

   `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 parameter 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 3-3 Sample 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 parameter.

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.

There are two ways that you can use the `ClasspathServlet`:

- To serve a resource from the system `CLASSPATH`, call the resource with a URL such as:

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

- 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: Since 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:comp/env`. See "Configuring Application-Scoped Resources" on page 3-14 for more information.

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.

This example assumes that you have defined a data source called `accountDataSource`. For more information, see *JDBC Data Sources Online Help*.

Listing 3-4 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>
```

Configuring Application-Scoped Resources

WebLogic Server binds application-scoped resource names to the application's local JNDI tree. The Web Application code accesses these resources by looking up the actual JNDI resource name relative to `java:comp/env`.

If your Web Application uses only application-scoped resources, you do not need to enter global JNDI resources names in the `weblogic.xml` deployment descriptor, as described in "Configuring External Resources" on page 3-13. (In fact, you can omit `weblogic.xml` entirely if you do not require any other features of that deployment descriptor.)

To configure application-scoped resources:

1. Enter the resource definition in the `weblogic-application.xml` deployment descriptor. See "weblogic-application.xml Deployment Descriptor Elements" in *Developing WebLogic Server Applications* for more information.

2. Ensure that Web Application code uses the same JNDI name specified in `weblogic-application.xml`, and that it references the name relative to the local JNDI tree at `java:comp/env`.

   Note: If Web Application code uses a different JNDI name to reference the resource, you must treat the resource as an external DataSource and configure the `weblogic.xml` deployment descriptor as illustrated in Listing 3-5.

Listing 3-5 Web Application Code Using a Different JNDI Name to Reference the Resource: Treated as External DataSource

```
Servlet code:
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup
                          ("java:comp/env/myDataSource");

weblogic-application.xml entries:

<weblogic-application>
   <data-source-name>myDataSource</data-source-name>
</weblogic-application>
```

Referencing EJBs in a Web Application

Web Application can access EJBs that are deployed as part of a different application (a different EAR file) by using an external reference, or they can be deployed within the scope of the Web Application as part of an EAR file. The procedures for referencing an EJB differ depending on whether the EJB is external or application-scoped.

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 "Invoking Deployed EJBs" in *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, JSP, or HTML page 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` elment 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, JSP, or HTML page 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, JSP, or HTML page) 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, JSP, or HTML page.

  Assuming the `ejb-ref-name` element is `ejb1`, the code in your servlet, JSP, or HTML page 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 an entry similar to the following:

Listing 3-8

```
<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.

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 codeset was used to encode the request.

There are two ways you can define the codeset:

- 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 codeset 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 a 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>
```

CHAPTER

4 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 4-2
- "Configuring Session Persistence" on page 4-4
- "Using URL Rewriting" on page 4-9

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. Tuning depends upon factors such as:

- How many users you expect to hit the servlet
- How many concurrent users 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.

HTTP Session Properties

You configure WebLogic Server session tracking with properties in the WebLogic-specific deployment descriptor, `weblogic.xml`.

For a complete list of session attributes, see "jsp-descriptor" on page B-9.

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 "jsp-descriptor" on page B-9 of the WebLogic-specific deployment descriptor, `weblogic.xml`. This value is set in seconds.
- Set the `<session-timeout>` (see "session-config" on page A-11) `element` in the Web Application deployment descriptor, `web.xml`.

Configuring Session Cookies

WebLogic Server uses cookies for session management when 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 lifetime is regarded as over. 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 "jsp-descriptor" on page B-9.

Using Cookies That Outlive a Session

For longer-lived client-side user data, your application should create and set its own cookies on the browser via the HTTP servlet API, and 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 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 assigned when the `request.getSession(true)` method is called. You should only make transient use of sessions.

You can set the maximum life of a cookie with the `CookieMaxAgeSecs` parameter 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 the *Programming WebLogic HTTP Servlets*.

Configuring Session Persistence

Use Session Persistence to permanently stored data from an HTTP session object in order 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.

Common Properties of Session Attributes

This section describes attributes common to file system, memory (single-server, non-replicated), JDBC, and cookie-based persistence. You can configure the number of sessions that are held in memory by setting the following properties in the `<session-descriptor>` 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 are expecting high volumes of simultaneous active sessions, you do not want these sessions to soak up the RAM of your server since 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 1024. The minimum is 16, and maximum is *`Integer.MAX_VALUE`*. An empty session uses less than 100 bytes, but grows as data is added to it.

`SwapIntervalSecs`

The interval the server waits between purging the least recently used sessions from the cache to the persistent store, when the `cacheEntries` limit has been reached.

If unset, this property defaults to 10 seconds; minimum is 1 second, and maximum is 604800 (1 week).

`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 parameter to a value less than the value set for the `<session-timeout>` element. Use this parameter to tune WebLogic Server for best performance on high traffic sites.

The minimum value is every second (1). The maximum value is once a week (604,800 seconds). If unset, the parameter defaults to 60 seconds.

To set `<session-timeout>`, see the "session-config" on page A-11 of the Web Application deployment descriptor `web.xml`.

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

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

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. Set the `PersistentStoreType` property in the `<session-descriptor>` element in the deployment descriptor file `weblogic.xml` to `file`.
2. Set the directory where WebLogic Server stores the sessions. See "`PersistentStoreDir`" on page B-7.

   If you do not explicitly set a value for this attribute, a temporary directory is created for you by WebLogic Server.

   If you are using file-based persistence in a cluster, you must explicitly set this attribute to a shared directory that is accessible to all the servers in a cluster. You must create this directory yourself.

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.

To configure JDBC-based persistent storage for sessions:

1. Set the `PersistentStoreType` property in the `<session-descriptor>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`, to `jdbc`.
2. Set a JDBC connection pool to be used for persistence storage with the `PersistentStorePool` property. Use the name of a connection pool that is defined in the WebLogic Server Administration Console.
3. Set an ACL for the connection that corresponds to the users that have permission.
4. 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. The following table shows the Column names and data types you should use when creating this table.

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

If you are using an Oracle DBMS, use the following SQL statement to create the `wl_servlet_sessions` table:

```
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) );
```

Modify the preceeding SQL statement for use with your DBMS.

Note: You can 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 with the `JDBConnectionTimeoutSecs` attribute. For more information, see "`JDBConnectionTimeoutSecs`" on page B-8.

Using Cookie-Based Session Persistence

Cookie-based session persistence provides a stateless solution for session persistence by storing all session data in a cookie that is stored 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 `IllegalArgument` 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.

To set up cookie-based session persistence:

1. In the `<session-descriptor>` element of `weblogic.xml`, set the `PersistentStoreType` parameter to `cookie`.
2. Optionally, set a name for the cookie using the `PersistentStoreCookieName` parameter. The default is `WLCOOKIE`.

Using URL Rewriting

In some situations, a browser or wireless device may not accept cookies, which makes session tracking using 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-descriptor>` element. The default value for this attribute is `true`. See "`URLRewritingEnabled`" on page B-8.

Coding Guidelines for URL Rewriting

There are some general guidelines for how your code should handle URLs in order to support 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 if the URL needs to be rewritten, and if so, it rewrites it by including the session ID in the URL. The session ID is appended to the URL and begins with 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 parameters), which limits the amount of data that can be transmitted using URL rewriting. To allow more space for parameters, you can limit the size of the session ID that is randomly generated by WebLogic Server. See "`IDLength`" on page B-8.

You can save additional space by setting the WAP Enabled attribute, which prevents WebLogic Server from sending primary/secondary information with the URL. You set the WAP Enabled attribute by selecting the Server > Configuration > HTTP tabs of the Administration Console.

CHAPTER

5 Application Events and Listeners

The following sections describe how to configure and use Web Application events and listeners:

- "Overview of Application Events and Listeners" on page 5-1
- "Servlet Context Events" on page 5-2
- "HTTP Session Events" on page 5-3
- "Configuring an Event Listener" on page 5-3
- "Writing a Listener Class" on page 5-4
- "Templates for Listener Classes" on page 5-5
- "Additional Resources" on page 5-6

Overview of Application Events and Listeners

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 Application event and listener classes in a Web Application.

For servlet context events, the event listener classes can receive notification when the Web Application is deployed or is being 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 events to:

- Manage database connections when a Web Application is deployed or shuts down
- Create counters
- 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 the event, and the methods invoked when the event occurs.

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

HTTP Session Events

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

Type of Event	Interface	Method
An HTTP session is activated.	`javax.servlet.http.HttpSessionListener`	`sessionCreated()`
An HTTP session is about to be passivated.	`javax.servlet.http.HttpSessionListener`	`sessionDestroyed()`
An attribute is added.	`javax.servlet.http.HttpSessionAttributesListener`	`attributeAdded()`
An attribute is removed.	`javax.servlet.http.HttpSessionAttributesListener`	`attributeRemoved()`
An attribute is replaced.	`javax.servlet.http.HttpSessionAttributesListener`	`attributeReplaced()`

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

To configure an event listener:

1. Open the `web.xml` deployment descriptor of the Web Application for which you are creating an event listener 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 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 listener class for each type of event. WebLogic Server invokes the event listeners 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</listen
er-class>
</listener>
```

3. Write and deploy the Listener class. See the next section, "Writing a Listener Class" on page 5-4, for details.

Writing a Listener Class

To write a listener class:

1. Create a new class that implements the appropriate interface for the type of event your class responds to. 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 Listener Classes" on page 5-5 for sample templates you can use to get started.
2. Create a public constructor that takes no arguments.
3. Implement the required methods of the interface. See the J2EE API Reference (Javadocs) 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 listener methods in a 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 Listener Classes

The following examples provide some basic templates for listener classes.

Servlet Context Listener Example

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

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 Listener Example

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

public final class mySessionAttributeListenerClass implements
   HttpSessionAttributesListener {

    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 attibute
         is replaced in a session.
      */
    }
}
```

Additional Resources

- *Servlet 2.3 Specification* from Sun Microsystems at `http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html`

- J2EE API Reference (Javadocs) at
  `http://java.sun.com/j2ee/tutorial/api/index.html`
- The J2EE Tutorial from Sun Microsystems: at
  `http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html`

CHAPTER

6 Configuring Security in Web Applications

The following sections describe how to configure security in Web Applications:

- "Overview of Configuring Security in Web Applications" on page 6-1
- "Setting Up Authentication for Web Applications" on page 6-2
- "Multiple Web Applications, Cookies, and Authentication" on page 6-4
- "Restricting Access to Resources in a Web Application" on page 6-5
- "Using Users and Roles Programmatically in Servlets" on page 6-6

To see overview, upgrade, and new information about WebLogic Server security, see *Programming WebLogic Security*.

Overview of Configuring Security in Web Applications

You can secure a Web Application by using authentication, by restricting access to certain resources in the Web Application, or by using security calls in your servlet code. At runtime, the WebLogic Server active security realm applies the Web application security constraints to the specified Web Application resources. For more information, see *Programming WebLogic Security*. Note that a security realm is shared across multiple virtual hosts.

Setting Up Authentication for Web Applications

To configure authentication for a Web Application, use the `<login-config>` element of the `web.xml` deployment descriptor. In this element you define the security realm containing the user credentials, the method of authentication, and the location of resources for authentication.

On application deployment, WebLogic Server reads role information from the `weblogic.xml` file. This information is used to populate the Authorization provider configured for the security realm. Once the role information is in the Authorization provider, changes made through the WebLogic Server Administration Console are not persisted to the `weblogic.xml` file.

Before you redeploy the application (which occurs when you redeploy it through the Console, modify it on disk, or restart WebLogic Server), you must enable the Ignore security data in deployment descriptors attribute on the Security Realm > General tab. Otherwise, the old data in the `weblogic.xml` file will overwrite any changes made using the WebLogic Server Administration Console.

To set up authentication for Web Applications:

1. Open the `web.xml` deployment descriptor in a text editor or using WebLogic Builder. For more information, see "Web Application Developer Tools" on page 1-7.
2. Specify the authentication method using the `<auth-method>` element. The available options are:

   `BASIC`
   : Basic authentication uses the Web Browser to display a username/password dialog box. This username and password is authenticated against the realm.

   `FORM`
   : Form-based authentication requires that you return an HTML page, JSP, or servlet form that renders a login screen. The fields returned from the form elements must be: `j_username` and `j_password`, and the action attribute must be `j_security_check`. Here is an example of the HTML coding for using FORM authentication:

Listing 6-1 Example FORM Authentication

```
<html>
       <head>
              <title>My Web Application</title>
       </head>
       <body>
              <p>Welcome to my Web application! Please log in </p>
              <form method="POST" action="j_secuirty_check">
              <p>Password: <input type="password"
              name="j_password"></p>
              <p><input type="submit" name="Submit"
              value="Login"></p>
              </form>
       </body>
</html>
```

The resource used to generate the HTML form may be an HTML page, a JSP, or a servlet. You define this resource with the `<form-login-page>` element.

The HTTP session object is created when the login page is served. Therefore, the `session.isNew()` method returns `FALSE` when called from pages served after successful authentication.

`CLIENT-CERT`

Uses client certificates to authenticate the request. For more information, see *Configuring the SSL Protocol*.

3. If you choose FORM authentication, also define the location of the resource used to generate the HTML page, JSP, or servlet with the `<form-login-page>` element and the resource that responds to a failed authentication with the `<form-error-page>` element. For instructions on configuring form authentication, see "form-login-config" on page A-20.
4. Specify a realm for authentication using the `<realm-name>` element. If you do not specify a particular realm, the realm defined with the Auth Realm Name field on the Web Application→Configuration→Other tab of the Administration Console is used. For more information, see "form-login-config" on page A-20.
5. If you want to define a separate login for a Web Application, see "Multiple Web Applications, Cookies, and Authentication" on page 6-4. Otherwise, all Web Applications that use the same cookie use a single sign-on for authentication.

Multiple Web Applications, Cookies, and Authentication

By default, WebLogic Server assigns the same cookie name (`JSESSIONID`) to all Web Applications. When you use any type of authentication, all Web Applications that use the same cookie name use a single sign-on for authentication. Once a user is authenticated, that authentication is valid for requests to any Web Application that uses the same cookie name. The user is not prompted again for authentication.

If you want to require separate authentication for a Web Application, you can specify a unique cookie name or cookie path for the Web Application. Specify the cookie name using the `CookieName` parameter and the cookie path with the `CookiePath` parameter, defined in the WebLogic-specific deployment descriptor `weblogic.xml`, in the `<session-descriptor>` element. For more information, see "jsp-descriptor" on page B-9.

If you want to retain the cookie name and still require independent authentication for each Web Application, you can set the cookie path parameter (`CookiePath`) differently for each Web Application.

Restricting Access to Resources in a Web Application

To restrict access to specified resources (servlets, JSPs, or HTML pages) in your Web Application, apply security constraints to those resources.

To configure a security constraint:

1. Open the `web.xml` and `weblogic.xml` deployment descriptors in a text editor or in the Administration Console. For more information, see "Web Application Developer Tools" on page 1-7.
2. In the WebLogic-specific deployment descriptor, `weblogic.xml`, define a role that is mapped to one or more principals in a security realm. Define roles with the "security-role" on page A-20. Then map these roles to principals in your realm with the "security-role-assignment" on page B-2.
3. In `web.xml`, define which resources in the Web Application the security constraint applies to by using the `<url-pattern>` element that is nested inside the `<web-resource-collection>` element. The `<url-pattern>` can refer to a directory, filename, or a `<servlet-mapping>`.

   Alternatively, to apply the security constraint to the entire Web Application, use the following entry:

   ```
   <url-pattern>/*</url-pattern>
   ```
4. In `web.xml`, define the HTTP method(s) (`GET` or `POST`) that the security constraint applies to by defining the `<http-method>` element that is nested inside the `<web-resource-collection>` element. Use separate `<http-method>` elements for each HTTP method.
5. In `web.xml`, define whether to use SSL for communication between client and server using the `<transport-guarantee>` element nested inside of the `<user-data-constraint>` method.

Listing 6-2 Sample Security Constraint

```
web.xml entries:
<security-constraint>
     <web-resource-collection>
          <web-resource-name>SecureOrdersEast</web-resource-name>
          <description>
             Security constraint for
             resources in the orders/east directory
          </description>
          <url-pattern>/orders/east/*</url-pattern>
          <http-method>POST</http-method>
          <http-method>GET</http-method>
     </web-resource-collection>
     <auth-constraint>
          <description>
           constraint for east coast sales
          </description>
          <role-name>east</role-name>
          <role-name>manager</role-name>
     </auth-constraint>
     <user-data-constraint>
          <description>SSL not required</description>
          <transport-guarantee>NONE</transport-guarantee>
     </user-data-constraint>
</security-constraint>

...
```

Using Users and Roles Programmatically in Servlets

You can write your servlets to access users and roles programmatically in your servlet code using the method `javax.servlet.http.HttpServletRequest.isUserInRole(String role)`. The string `role` is mapped to the name supplied in the `<role-name>` element nested inside the `<security-role-ref>` element of a `<servlet>` declaration in the Web

Application deployment descriptor. The `<role-link>` element maps to a `<role-name>` defined in the `<security-role>` element of the Web Application deployment descriptor.

The following listing provides an example.

Listing 6-3 Example of Security Role Mapping

```
Servlet code:
isUserInRole("manager");

web.xml entries:

<servlet>
. . .
   <role-name>manager</role-name>
   <role-link>mgr</role-link>
. . .
</servlet>

<security-role>
   <role-name>mgr</role-name>
</security-role>

weblogic.xml entries:

<security-role-assignment>
   <role-name>mgr</role-name>
   <principal-name>al</principal-name>
   <principal-name>george</principal-name>
   <principal-name>ralph</principal-name>
</security-role-ref>
```

CHAPTER

7 Filters

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

- "Overview of Filters" on page 7-1
- "Configuring Filters" on page 7-3
- "Writing a Filter" on page 7-5
- "Example of a Filter Class" on page 7-7
- "Filtering the Servlet Response Object" on page 7-8
- "Additional Resources" on page 7-8

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.ServletRequest` and `javax.servlet.ServletResponse` 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.

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.)

Configuring Filters

You configure filters as part of a Web Application, using the application's `web.xml` deployment descriptor. In the deployment descriptor, you declare the filter and then map the filter to a URL pattern or to a specific servlet in the Web Application. You can declare 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 1-7. 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:

   ```
   <filter>
     <icon>
       <small-icon>MySmallIcon.gif</small-icon>
       <large-icon>MyLargeIcon.gif</large-icon>
     </icon>
     <filter-name>myFilter1</filter-name>
     <display-name>filter 1</display-name>
     <description>This is my filter</description>
     <filter-class>examples.myFilterClass</filter-class>
   </filter>
   ```

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

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

   ```
   <filter>
     <icon>
       <small-icon>MySmallIcon.gif</small-icon>
       <large-icon>MyLargeIcon.gif</large-icon>
   ```

```
  </icon>
  <filter-name>myFilter1</filter-name>
  <display-name>filter 1</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 parameters 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 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-hame>myServlet</servlet-name>
     </filter-napping>
     ```

5. To create a chain of filters, specify multiple filter mappings. For more information, see "Configuring a Chain of Filters" on page 7-5.

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` 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` 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.

Writing a Filter

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:

`doFilter()`
: Use this 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.

`GetFilterConfig()`
: Use this method to gain access to the javax.servlet.FilterConfig (see `http://java.sun.com/j2ee/tutorial/api/javax/servlet/FilterConfig.html`) object.

`setFilterConfig()`
: Use this method to set the javax.servlet.FilterConfig (see `http://java.sun.com/j2ee/tutorial/api/javax/servlet/FilterConfig.html`) object.

Several other methods are available on the `FilterConfig` object for accessing the name of the filter, the `ServletContext` and the filter's initialization parameters. 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.

Example of a Filter Class

The following code example demonstrates the basic structure of a `Filter` class.

Listing 7-1 Filter Class Example

```
import javax.servlet.*;
public class Filter1Impl implements Filter
{
    private FilterConfig filterConfig;

    public void doFilter(ServletRequest req,
        ServletResponse res, FilterChain fc)
        throws java.io.IOException, javax.servlet.ServletException
    {
      // Execute a task such as logging.
      //...

      fc.doFilter(req,res); // invoke next item in the chain --
                            // either another filter or the
                            // originally requested resource.

    }

    public FilterConfig getFilterConfig()
    {
      // Execute tasks
      return filterConfig;
    }

    public void setFilterConfig(FilterConfig cfg)
    {
      // Execute tasks
      filterConfig = cfg;
    }
}
```

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 at `http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html`
- J2EE API Reference (Javadocs) at `http://java.sun.com/j2ee/tutorial/api/index.html`
- The J2EE Tutorial from Sun Microsystems at `http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html`

CHAPTER

A web.xml Deployment Descriptor Elements

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

- "icon" on page A-2
- "display-name" on page A-3
- "description" on page A-3
- "distributable" on page A-3
- "context-param" on page A-4
- "filter" on page A-4
- "filter-mapping" on page A-5
- "listener" on page A-6
- "servlet" on page A-6
- "servlet-mapping" on page A-10
- "session-config" on page A-11
- "mime-mapping" on page A-11
- "welcome-file-list" on page A-12
- "error-page" on page A-13
- "taglib" on page A-13

- "resource-env-ref" on page A-14
- "resource-ref" on page A-15
- "security-constraint" on page A-16
- "login-config" on page A-19
- "security-role" on page A-20
- "env-entry" on page A-21
- "ejb-ref" on page A-21
- "ejb-local-ref" on page A-22

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.)

This element is not currently used by WebLogic Server.

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

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

display-name

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

Example usage:

```
<context-param>
        <param-name>weblogic.httpd.inputCharset./*</param-name>
        <param-value>UTF-8</param-value>
</context-param>
```

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

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

filter

The filter element defines a filter class and its initialization parameters. For more information on filters, see "Configuring Filters" on page 7-3.

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

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

filter-mapping

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

Element	Required/ Optional	Description
`<filter-name>`	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 `<filter>` element with the `<filter-name>` element.

Element	Required/ Optional	Description
<url-pattern>	Required - or map by <servlet>	Describes a pattern used to resolve URLs. The portion of the URL after the http://host:port + ContextPath is compared to the <url-pattern> by WebLogic Server. If the patterns match, the filter mapped in this element is called. Example patterns: /soda/grape/* /foo/* /contents *.foo The URL must follow the rules specified in the Servlet 2.3 Specification.
<servlet>	Required - or map by <url-pattern>	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
<listener-class>	Optional	Name of the class that responds to a Web Application event.

For more information, see "Configuring an Event Listener" on page 5-3.

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

icon

This is an element within the "servlet" on page A-6.

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

init-param

This is an element within the "servlet" on page A-6.

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

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

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

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

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

WebLogic Server recognizes the special initialization parameter, `wl-dispatch-policy`, to assign a servlet or JSP to an available execute queue. For example, the following example assigns a servlet to use the execute threads available in an execute queue named `CriticalWebApp`:

```
<servlet>
   ...
   <init-param>
      <param-name>wl-dispatch-policy</param-name>
      <param-value>CriticalWebApp</param-value>
   </init-param>
</servlet>
```

If the `CriticalWebApp` queue is not available, the servlet will use execute threads available in the default WebLogic Server execute queue. See Setting Thread Count for more information about configuring execute threads in WebLogic Server. See Using Execute Queues to Control Thread Usage for more information about creating and using queues.

security-role-ref

This is an element within the "servlet" on page A-6.

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

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

Element	Required/ Optional	Description
`<description>`	Optional	Text description of the role.

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

servlet-mapping

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

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

Element	Required/ Optional	Description
<servlet-name>	Required	The name of the servlet to which you are mapping a URL pattern. This name corresponds to the name you assigned a servlet in a <servlet> declaration tag.
<url-pattern>	Required	Describes a pattern used to resolve URLs. The portion of the URL after the http://host:port + WebAppName is compared to the <url-pattern> by WebLogic Server. If the patterns match, the servlet mapped in this element will be called. Example patterns: /soda/grape/* /foo/* /contents *.foo The URL must follow the rules specified in the Servlet 2.2 Specification. For additional examples of servlet mapping, see "Servlet Mapping" on page 3-2.

session-config

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

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

Element	Required/ Optional	Description
`<session-timeout>`	Optional	The number of minutes after which sessions in this Web Application expire. The value set in this element overrides the value set in the `TimeoutSecs` parameter of the `<session-descriptor>` element in the WebLogic-specific deployment descriptor `weblogic.xml`, unless one of the special values listed here is entered. Default value: -2 Maximum value: Integer.MAX_VALUE ÷ 60 Special values: ■ -2 = Use the value set by `TimeoutSecs` in `<session-descriptor>` element of `weblogic.xml` ■ -1 = Sessions do not timeout. The value set in `<session-descriptor>` element of `weblogic.xml` is ignored. For more information, see "jsp-descriptor" on page B-9.

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

welcome-file-list

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

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

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
`<taglib-location>`	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 `WEB-INF` directory so it is not publicly available over an HTTP request.
`<taglib-uri>`	Required	Describes a URI, relative to the location of the web.xml document, identifying a Tag Library used in the Web Application. If the URI matches the URI string used in the taglib directive on the JSP page, this taglib is used.

resource-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
`<description>`	Optional	Provides a description of the resource environment reference.
`<resource-env-ref -name>`	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 `java:comp/env` context and must be unique within a Web application.
`<resource-env-ref -type>`	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
`<description>`	Optional	A text description.
`<res-ref-name>`	Required	The name of the resource used in the JNDI tree. Servlets in the Web Application use this name to look up a reference to the resource.
`<res-type>`	Required	The Java type of the resource that corresponds to the reference name. Use the full package name of the Java type.

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: ■ 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 more information, see "Configuring Security in Web Applications" on page 6-1.

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
<web-resource-name>	Required	The name of this Web resource collection.
<description>	Optional	A text description of this security constraint.
<url-pattern>	Optional	Use one or more of the <url-pattern> elements to declare to which URL patterns this security constraint applies. If you do not use at least one of these elements, this <web-resource-collection> is ignored by WebLogic Server.
<http-method>	Optional	Use one or more of the <http-method> elements to declare which HTTP methods (usually, GET or POST) are subject to the authorization constraint. If you omit the <http-method> 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
`<description>`	Optional	A text description of this security constraint.
`<role-name>`	Optional	Defines which security roles can access resources defined in this security-constraint. Security role names are mapped to principals using the `security-role-ref`. See "security-role-ref" on page A-9.

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
`<description>`	Optional	A text description.
`<transport-guarantee>`	Required	Specifies that the communication between client and server. WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the `INTEGRAL` or `CONFIDENTIAL` transport guarantee. Range of values: ■ `NONE`–The application does not require any transport guarantees. ■ `INTEGRAL`–The application requires that the data be sent between the client and server in such a way that it cannot be changed in transit. ■ `CONFIDENTIAL`–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
`<auth-method>`	Optional	Specifies the method used to authenticate the user. Possible values: `BASIC` - uses browser authentication `FORM` - uses a user-written HTML form `CLIENT-CERT`
`<realm-name>`	Optional	The name of the realm that is referenced to authenticate the user credentials. If omitted, the realm defined with the Auth Realm Name field on the Web Application→Configuration→Other tab of the Administration Console is used by default. *For more information, see* "Managing WebLogic Security". **Note:** The `<realm-name>` element does not refer to security realms within WebLogic Server. This element defines the realm name to use in HTTP Basic authorization. **Note:** 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.
`<form-login-config>`	Optional	Use this element if you configure the `<auth-method>` to FORM. See "form-login-config" on page A-20.

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
`<form-login-page>`	Required	The URI of a Web resource relative to the document root, used to authenticate the user. This can be an HTML page, JSP, or HTTP servlet, and must return an HTML page containing a FORM that conforms to a specific naming convention. For more information, see "Setting Up Authentication for Web Applications" on page 6-2.
`<form-error-page>`	Required	The URI of a Web resource relative to the document root, sent to the user in response to a failed authentication login.

security-role

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

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

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

ejb-ref

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
`<description>`	Optional	A text description of the reference.
`<ejb-ref-name>`	Required	The name of the EJB used in the Web Application. This name is mapped to the JNDI tree in the WebLogic-specific deployment descriptor `weblogic.xml`. For more information, see "ejb-reference-description" on page B-4.
`<ejb-ref-type>`	Required	The expected Java class type of the referenced EJB.
`<home>`	Required	The fully qualified class name of the EJB home interface.
`<remote>`	Required	The fully qualified class name of the EJB remote interface.
`<ejb-link>`	Optional	The `<ejb-name>` of an EJB in an encompassing J2EE application package.
<run-as>	Optional	A security role whose security context is applied to the referenced EJB. Must be a security role defined with the `<security-role>` 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's referencing 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
`<description>`	Optional	A text description of the reference.
`<ejb-ref-name>`	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 `java:comp/env` context. The name must be unique within the Web application. It is recommended that name is prefixed with `ejb/`. For example: `<ejb-ref-name>ejb/Payroll</ejb-ref-name>`

`<ejb-ref-type>`	Required	The `ejb-ref-type` element contains the expected type of the referenced enterprise bean. The `ejb-ref-type` element must be one of the following:
		`<ejb-ref-type>Entity</ejb-ref-type>`
		`<ejb-ref-type>Session</ejb-ref-type>`

`<local-home>`	Required	Contains the fully-qualified name of the enterprise bean's local home interface.
`<local>`	Required	Contains the fully-qualified name of the enterprise bean's local interface.

Element	Required/ Optional	Description
<ejb-link>	Optional	The ejb-link element is used in the ejb-ref or ejb-local-ref elements to specify that an EJB reference is linked to an enterprise bean. The name in the ejb-link element is composed of a path name specifying the ejb-jar containing the referenced enterprise bean with the ejb-name of the target bean appended and separated from the path name by "#". The path name is relative to the war file containing the web application that is referencing the enterprise bean. This allows multiple enterprise beans with the same ejb-name to be uniquely identified. Used in: ejb-local-ref, ejb-ref Examples: <ejb-link>EmployeeRecord</ejb-link> <ejb-link>../products/product.jar#ProductEJB</ejb-link>

CHAPTER

B weblogic.xml Deployment Descriptor Elements

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

- "description" on page B-2
- "weblogic-version" on page B-2
- "security-role-assignment" on page B-2
- "reference-descriptor" on page B-3
- "session-descriptor" on page B-4
- "jsp-descriptor" on page B-9
- "auth-filter" on page B-11
- "container-descriptor" on page B-11
- "charset-params" on page B-13
- "virtual-directory-mapping" on page B-14
- "url-match-map" on page B-15
- "preprocessor" on page B-16

- "preprocessor-mapping" on page B-16
- "security-permission" on page B-17
- "context-root" on page B-17
- "wl-dispatch-policy" on page B-18
- "init-as" on page B-19
- "destroy-as" on page B-19
- "index-directory" on page B-19

You can also access the Document Type Descriptor (DTD) for `weblogic.xml` at http://www.bea.com/servers/wls710/dtd/weblogic710-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
`<role-name>`	Required	Specifies the name of a security role.
`<principal-name>`	Required	Specifies the name of a principal that is defined in the security realm. You can use multiple `<principal-name>` elements to map principals to a role. For more information on security realms, see *Programming WebLogic Security*.

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-description

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

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

ejb-reference-description

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

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

session-descriptor

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

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

```
      </param-value>
    </session-param>
</session-descriptor>
```

Session Parameter Names and Values

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

Parameter Name	Default Value	Parameter Value
`CookieDomain`	Null	Specifies the domain for which the cookie is valid. For example, setting `CookieDomain` to `.mydomain.com` returns cookies to any server in the `*.mydomain.com` domain. The domain name must have at least two components. Setting a name to `*.com` or `*.net` is not valid. If unset, this parameter defaults to the server that issued the cookie. For more information, see `Cookie.setDomain()` in the Servlet specification from Sun Microsystems.
`CookieComment`	Weblogic Server Session Tracking Cookie	Specifies the comment that identifies the session tracking cookie in the cookie file. If unset, this parameter defaults to `WebLogic Session Tracking Cookie`. You may provide a more specific name for your application.
`CookieMaxAgeSecs`	-1	Sets the life span of the session cookie, in seconds, after which it expires on the client. If the value is 0, the cookie expires immediately. The maximum value is `Integer.MAX_VALUE`, where the cookie lasts forever. If set to `-1`, the cookie expires when the user exits the browser. For more information about cookies, see "Using Sessions and Session Persistence in Web Applications" on page 4-1.

Parameter Name	Default Value	Parameter Value
CookieName	JSESSIONID	Defines the session cookie name. Defaults to JSESSIONID if unset. You may set this to a more specific name for your application.
CookiePath	Null	Specifies the pathname to which the browser sends cookies. If unset, this parameter defaults to / (slash), where the browser sends cookies to all URLs served by WebLogic Server. You may set the path to a narrower mapping, to limit the request URLs to which the browser sends cookies.
CookiesEnabled	true	Use of session cookies is enabled by default and is recommended, but you can disable them by setting this property to false. You might turn this option off to test.
InvalidationIntervalSecs	60	Sets the time, in seconds, that WebLogic Server waits between doing house-cleaning checks for timed-out and invalid sessions, and deleting the old sessions and freeing up memory. Use this parameter to tune WebLogic Server for best performance on high traffic sites. The minimum value is every second (1). The maximum value is once a week (604,800 seconds). If unset, the parameter defaults to 60 seconds.

Parameter Name	Default Value	Parameter Value
`PersistentStoreDir`	`session_db`	If you have set `PersistentStoreType` to `file`, this parameter sets the directory path where WebLogic Server will store the sessions. The directory path is either relative to the temp directory or an absolute path. The temp directory is either a generated directory under the `WEB-INF` directory of the Web Application, or a directory specified by the context-param `javax.servlet.context.tmpdir`. Ensure that you have enough disk space to store the *number of valid sessions* multiplied by the *size of each session*. You can find the size of a session by looking at the files created in the `PersistentStoreDir`. Note that the size of each session can vary as the size of serialized session data changes. You can make file-persistent sessions clusterable by making this directory a shared directory among different servers. You must create this directory manually.
`PersistentStorePool`	None	Specifies the name of a JDBC connection pool to be used for persistence storage.
`PersistentStoreType`	`memory`	Sets the persistent store method to one of the following options: ■ `memory`—Disables persistent session storage. ■ `file`—Uses file-based persistence (See also `PersistentStoreDir`, above). ■ `jdbc`—Uses a database to store persistent sessions. (see also `PersistentStorePool`, above). ■ `replicated`—Same as `memory`, but session data is replicated across the clustered servers. ■ `cookie`—All session data is stored in a cookie in the user's browser.
`PersistentStoreCookieName`	`WLCOOKIE`	Sets the name of the cookie used for cookie-based persistence. For more information, see "Using Cookie-Based Session Persistence" on page 4-8.

Parameter Name	Default Value	Parameter Value
`IDLength`	52	Sets the size of the session ID. The minimum value is 8 bytes and the maximum value is `Integer.MAX_VALUE`. If you are writing a WAP application, you must use URL rewriting because the WAP protocol does not support cookies. Also, some WAP devices have a 128-character limit on URL length (including parameters), which limits the amount of data that can be transmitted using URL re-writing. To allow more space for parameters, use this parameter to limit the size of the session ID that is randomly generated by WebLogic Server.
`TimeoutSecs`	3600	Sets the time, in seconds, that WebLogic Server waits before timing out a session, where x is the number of seconds between a session's activity. Minimum value is 1, default is 3600, and maximum value is integer MAX_VALUE. On busy sites, you can tune your application by adjusting the timeout of sessions. While you want to give a browser client every opportunity to finish a session, you do not want to tie up the server needlessly if the user has left the site or otherwise abandoned the session. This parameter can be overridden by the `session-timeout` element (defined in minutes) in `web.xml`. For more information, see "session-config" on page A-11.
`JDBConnectionTimeoutSecs`	120	Sets the time, in seconds, that WebLogic Server waits before timing out a JDBC connection, where x is the number of seconds between.
`URLRewritingEnabled`	true	Enables URL rewriting, which encodes the session ID into the URL and provides session tracking if cookies are disabled in the browser.
`ConsoleMainAttribute`		If you enable Session Monitoring in the WebLogic Server Administration Console, set this parameter to the name of the session parameter you will use to identify each session that is monitored.

jsp-descriptor

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

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

JSP Parameter Names and Values

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

Parameter Name	Default Value	Parameter Value

`compileCommand`	`javac`, or the Java compiler defined for a server under the configuration /tuning tab of the WebLogic Server Administration Console	Specifies the full pathname of the standard Java compiler used to compile the generated JSP servlets. For example, to use the standard Java compiler, specify its location on your system as shown below:
		`<param-value>`
		`/jdk130/bin/javac.exe`
		`</param-value>`
		For faster performance, specify a different compiler, such as IBM Jikes or Symantec `sj`.

Parameter Name	Default Value	Parameter Value
compileFlags	None	Passes one or more command-line flags to the compiler. Enclose multiple flags in quotes, separated by a space. For example: <jsp-param> <param-name>compileFlags</param-name> <param-value>"-g -v"</param-value> </jsp-param>
compilerclass	None	Name of a Java compiler that is executed in WebLogic Servers's virtual machine. (Used in place of an executable compiler such as javac or sj.) If this parameter is set, the compileCommand parameter is ignored.
debug	None	When set to true this adds JSP line numbers to generated class files to aid debugging.
encoding	Default encoding of your platform	Specifies the default character set used in the JSP page. Use standard Java character set names (see http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.htm). If not set, this parameter defaults to the encoding for your platform. A JSP page directive (included in the JSP code) overrides this setting. For example: <%@ page contentType="text/html; charset=custom-encoding"%>
compilerSupports Encoding	true	When set to true, the JSP compiler uses the encoding specified with the contentType attribute contained in the page directive on the JSP page, or, if a contentType is not specified, the encoding defined with the encoding parameter in the jsp-descriptor. When set to false, the JSP compiler uses the default encoding for the JVM when creating the intermediate .java file.
exactMapping	true	When true, upon the first request for a JSP the newly created JspStub is mapped to the exact request. If exactMapping is set to false the webapp container generates non-exact url mapping for JSPs. exactMapping 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 parameter is set to true, the intermediate Java files are deleted after they are compiled.

Parameter Name	Default Value	Parameter Value
`noTryBlocks`	false	If a JSP file has numerous or deeply nested custom JSP tags and you receive a `java.lang.VerifyError` exception when compiling, use this flag to allow the JSPs to compile correctly.
`packagePrefix`	`jsp_servlet`	Specifies the package into which all JSP pages are compiled.
`pageCheckSeconds`	`1`	Sets the interval, in seconds, at which WebLogic Server checks to see if JSP files have changed and need recompiling. Dependencies are also checked and recursively reloaded if changed. If set to 0, pages are checked on every request. If set to `-1`, page checking and recompiling is disabled.
precompile	false	When set to true, WebLogic Server automatically precompiles all JSPs when the Web Application is deployed or re-deployed or when starting WebLogic Server.
`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.

auth-filter

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

container-descriptor

The `<container-descriptor>` element defines general parameters 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-content-type

If the redirect-content-type element is set, then the servlet container sets that type on the response for internal redirects (for example, for welcome files).

redirect-content

If the redirect-content element is set, then the servlet container will use that as the value for the user readable data used in a redirect.

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.

charset-params

The <charset-params> Element is used to define codeset behavior for non-unicode operations.

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 "Determining the Encoding of an HTTP Request" on page 3-20.

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

Element	Required/ Optional	Description
<resource-path>	Required	A path which, if included in the URL of a request, signals WebLogic Server to use the Java character set specified by <java-charset-name>.
<java-charset-name>	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 3-21.

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

Element	Required/ Optional	Description
`<iana-charset-name>`	Required	Specifies the IANA character set name that is to be mapped to the Java character set specified by the `<java-charset-name>` element.
`<java-charset-name>`	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
`<local-path>`	Required	Specifies a physical location on the disk.
<url-pattern>	Required	Contains the URL pattern of the mapping. Must follow the rules specified in Section 11.2 of the Servlet API Specification.

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-app is deployed standalone, `application.xml` won't 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.xmll`, 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`.
4. When subsequent Web Applications have context root names that would duplicate a context root name already in use, a number is appended to the would-be duplicates. For instance if `MyWebApp` is already in use, another Web Application whose context root would be named `MyWebApp` is instead called `MyWebApp-1`, to be followed if necessary by `MyWebApp-2,` and so on.

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. This Web application level param can be overridden at the individual servlet/jsp level. For example:

```
<servlet>
  ...
  <init-param>
    <param-name>wl-dispatch-policy</param-name>
    <param-value>CriticalAppQueue</param-value>
  </init-param>
```

```
</servlet>
```

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-directory

This sorts the directory listing by file name, size, or date last modified.

For example:

```
<weblogic-web-app>
     <index-directory-enabled>true</index-directory-enabled>
     <index-directory-sort-by>SIZE</index-directory-sort-by>
</weblogic-web-app>
```

Index

A

application events 5-1
appliction event listeners 5-1
authentication
    and multiple web applications,and cookies 6-4
    basic 6-2
    client certificates 6-3
    form-based 6-2

C

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

D

default servlet 3-8
deploying
    Web Application 1-5
deployment
    in an Enterprise Application 2-6
    overview 2-1
deployment descriptor
    re-deployment 2-4
directory structure 1-6
document root 1-6
documentation, where to find it x
doFilter() 7-5

E

ear 2-6
Enterprise Application
    deploying a Web Application in 2-6
error pages 3-9
event listener
    declaration 5-4
event listeners
    configuring 5-3
events
    declaration 5-4
exploded directory format
    re-deployment 2-2

F

filter class 7-5
filter mapping 7-4
    to a servlet 7-4
    URL pattern 7-4
filters
    and Web Applications 7-2
    chaining 7-5
    configuring 7-3
    declaration 7-3

mappings 7-4
overview 7-1
uses 7-2
writing a filter class 7-5

G

GetFilterConfig() 7-5

H

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

I

index-directory B-19
init params 3-4
in-memory replication 4-4

J

JSP
modifying 2-4
refreshing 2-4
tag libraries 3-6

L

listener
writing a listener class 5-4
listener class 5-4
listeners 5-1
configuring 5-3
HTTP session events 5-3
servlet context events 5-2

M

mapping
filters 7-4
modifying components 2-4
modifying JSP 2-4

P

persistence for sessions 4-4
printing product documentation x

R

REDEPLOY file 2-2
re-deployment 2-2
.war archive 2-2
and HTTP sessions 2-4
exploded directory format 2-2
of Java classes 2-4
using administration console 2-3
using REDEPLOY file 2-2
when using auto-deployment 2-2
refreshing
JSP 2-4
response 7-1

S

security
applying programatically in servlet 6-6
authentication 6-2
client certificates 6-3
constraints 6-5
Web Applications 6-1
servlet
configuration 3-2
default servlet 3-8
initialization parameters 3-4
mapping 3-2
url-pattern 3-2
servlet context events 5-2
servlets
compiling into class files 1-4
session persistence

file-based 4-6
JDBC (database) 4-7
single server 4-6
Session Timeout 4-2
sessions 4-2
cookies 4-3
persistence 4-4
Session Timeout attribute 4-2
setting up 4-2
URL rewriting 4-9
URL rewriting and WAP 4-10
setFilterConfig 7-5
support
technical xi

U

URL rewriting 4-9

W

WAP 4-10
Web Application
configuring external resources 3-13
default servlet 3-8
deploying 1-5
directory structure 1-6
error page 3-9
security 6-1
security constraint 6-5
URI 1-7
Web applications
compiling servlets into class files 1-4
WEB-INF directory 1-6
welcome pages 3-7