BEA WebLogic Server® and WebLogic Express™

Developing Web Applications, Servlets, and JSPs for WebLogic Server

Version 9.2
Revised: June 27, 2007

Contents

1. Introduction and Roadmap

2. Understanding Web Applications, Servlets, and JSPs

3. Creating and Configuring Web Applications

4. Creating and Configuring Servlets

5. Creating and Configuring JSPs

6. Configuring Resources in a Web Application

7. Servlet Programming Tasks

8. Using Sessions and Session Persistence

11.Filters

12.Using WebLogic JSP Form Validation Tags

B. weblogic.xml Deployment Descriptor Elements

C. Web Application Best Practices

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Developing Web Applications, Servlets, and JSPs for WebLogic Server*

Document Scope and Audience

This document is a resource for software developers who develop Web applications and components such as HTTP servlets and JavaServer Pages (JSPs) for deployment on WebLogic Server®. This document is also a resource for Web application users and deployers. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Server Web applications for a particular application.

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

This document does not address production phase administration, monitoring, or performance tuning topics. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-3.

It is assumed that the reader is familiar with J2EE and Web application concepts. This document emphasizes the value-added features provided by WebLogic Server Web applications and key information about how to use WebLogic Server features and facilities to get a Web application up and running.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.
- Chapter 2, "Understanding Web Applications, Servlets, and JSPs," provides an overview of WebLogic Server Web applications servlets, and Java Server Pages (JSPs).
- Chapter 3, "Creating and Configuring Web Applications," describes how to create and configure Web application resources.
- Chapter 4, "Creating and Configuring Servlets," describes how to create and configure servlets.
- Chapter 5, "Creating and Configuring JSPs," describes how to create and configure JSPs.
- Chapter 6, "Configuring Resources in a Web Application," describes how to configure Web application resources.
- Chapter 7, "Servlet Programming Tasks," describes how to write HTTP servlets in a WebLogic Server environment.
- Chapter 8, "Using Sessions and Session Persistence," describes how to set up sessions and session persistence.
- Chapter 9, "Application Events and Event Listener Classes," discusses application events and event listener classes.
- Chapter 11, "Filters," provides information about using filters in a Web application.
- Chapter 10, "WebLogic JSP Reference," provides reference information for writing JavaServer Pages (JSPs).
- Chapter 12, "Using WebLogic JSP Form Validation Tags," describes how to use WebLogic JSP form validation tags.

- Chapter 13, "Using Custom WebLogic JSP Tags (cache, process, repeat)," describes the use of three custom JSP tags—`cache`, `repeat`, and `process`—provided with the WebLogic Server distribution.
- Chapter 14, "Using the WebLogic EJB to JSP Integration Tool," describes how to use the WebLogic EJB-to-JSP integration tool to create JSP tag libraries that you can use to invoke EJBs in a JavaServer Page (JSP). This document assumes at least some familiarity with both EJB and JSP.
- Appendix A, "web.xml Deployment Descriptor Elements," describes the deployment descriptor elements defined in the `web.xml` schema under the root element `<web-app>`.
- Appendix B, "weblogic.xml Deployment Descriptor Elements," provides a complete reference for the schema for the WebLogic Server-specific deployment descriptor `weblogic.xml`.
- Appendix C, "Web Application Best Practices," contains BEA best practices for designing, developing, and deploying WebLogic Web applications and application resources.

Related Documentation

This document contains Web application-specific design and development information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Developing Applications with WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- *Upgrading WebLogic Application Environments* contains information about Web Applications, JSP, and Servlet compatibility with previous WebLogic Server releases.
- JavaServer Pages Tutorial from Sun Microsystems at `http://java.sun.com/products/jsp/docs.html`
- JSP product overview from Sun Microsystems at `http://www.java.sun.com/products/jsp/index.html`
- JSP 2.0 Specification from Sun Microsystems at `http://java.sun.com/products/jsp/download.html`

- Servlet 2.4 Specification from Sun Microsystems at `http://java.sun.com/products/servlet/download.html#specs`
- 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/`

Examples for the Web Application Developer

In addition to this document, BEA Systems provides examples for software developers within the context of the Avitek Medical Records Application (MedRec) sample, discussed in the next section.

Avitek Medical Records Application (MedRec)

MedRec is an end-to-end sample J2EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and J2EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

Web Application Examples in the WebLogic Server Distribution

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

BEA provides several Web application, servlet, and JSP examples with this release of WebLogic Server. BEA recommends that you run these Web application examples before developing your own Web applications.

New and Changed Features In This Release

For this manual, WebLogic Server introduces the following improvements for version 9.2:

Note: WebLogic Server changed substantially in version 9.0, and these changes apply to later releases as well.

- For a detailed description of features and functionality introduced in WebLogic Server 9.0, see What's New in WebLogic Server 9.0.
- For a description of new and changed functionality in WebLogic Server 9.1, see What's New in WebLogic Server 9.1.

WebApp Libraries

Just as standard shared J2EE applications can be deployed to WebLogic Server as application-libraries, a standard Web application can be deployed to WebLogic Server as a webapp-library so that other Web applications can refer to these libraries. For information on referencing these WebApp libraries with your Web applications, see Using WebApp Libraries With Web Applications in *Developing Applications with WebLogic Server*.

JSF and JSTL Libraries

Three JSF (JavaServer™ Faces) and JSTL (JSP™ Standard Tag Library) packages are bundled with WebLogic Server as WebApp libraries. These libraries can be referenced by standard Web applications that use JSF or JSTL functionality.

The following three packages are being made available as WebApp libraries in release 9.2:

- MyFaces JSF library – http://myfaces.apache.org
- Sun JSF RI library – https://javaserverfaces.dev.java.net

- JSTL library – http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

See "Using JSF and JSTL With Web Applications" on page 3-8.

Using Two-Way SSL With Generic Proxy Servlets

When using generic proxy servlets, you can define the `KeyStore` initialization parameters to use two-way SSL with your own identity certificate and key. For more information, refer to the following documents:

- "Proxying Requests to Another Web Server" on page 7-20
- Configuring Proxy Plug-Ins in *Developing Applications With WebLogic Server*
- Setting Up a Proxy to a Secondary Web Server in *Using Web Server Plug-Ins with WebLogic Server*

Servlet Authentication Fallback Mechanism

The Servlet 2.4 specification allows you to define the authentication method (BASIC, FORM, etc.) to be used in a Web application. WebLogic Server 9.2 provides an `auth-method` security module that allows you to define multiple authentication methods (as a comma separated list), so the container can provide a fallback mechanism. Authentication will be attempted in the order the values are defined in the `auth-method` list.

See Providing a Fallback Mechanism for Authentication Methods in *Programming WebLogic Security*.

Future Response Model for HTTP Servlets

In general, WebLogic Server processes incoming HTTP requests and the response is returned immediately to the client. Such connections are handled synchronously by the same thread; however, some HTTP requests may require longer processing time. Handling these requests synchronously causes the thread to be held, waiting until the request is processed and the response sent. To avoid this scenario, WebLogic Server provides two classes that handle HTTP requests asynchronously by de-coupling the response from the thread that handles the incoming request.

See "A Future Response Model for HTTP Servlets" on page 7-25.

Deprecated Features

The `docHome` parameter for FileServlet has been deprecated.Use virtual directories as an alternative.

CHAPTER 2

Understanding Web Applications, Servlets, and JSPs

The following sections provide an overview of WebLogic Server Web applications, servlets, and Java Server Pages (JSPs):

The Web Applications Container

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 adds service-refs (Web services) and message-destination-refs (JMS destinations/queues) to an application. It can also define links to outside resources such as Enterprise JavaBeans (EJBs).

Web applications deployed on WebLogic Server use a standard J2EE deployment descriptor file and a WebLogic-specific deployment descriptor file to define their resources and operating attributes.

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

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

- One servlet or Java Server Page (JSP), along with any helper classes.
- A `web.xml` deployment descriptor, which is a J2EE standard XML document that describes the contents of a WAR file.
- A `weblogic.xml` deployment descriptor, which is an XML document containing WebLogic Server-specific elements for Web applications.
- A WAR file can also include HTML or XML pages and supporting files such as image and multimedia files.

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

BEA recommends that you package and deploy your stand-alone Web applications as part of an Enterprise application. This is a BEA best practice, which allows for easier application migration, additions, and changes. Also, packaging your applications as part of an Enterprise application allows you to take advantage of the split development directory structure, which provides a number of benefits over the traditional single directory structure.

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 Developer Tools" on page 2-6.

Servlets

A servlet is a Java class that runs in a Java-enabled server. An HTTP servlet is a special type of servlet that handles an HTTP request and provides an HTTP response, usually in the form of an HTML page. The most common use of WebLogic HTTP Servlets is to create interactive applications using standard Web browsers for the client-side presentation while WebLogic Server handles the business logic as a server-side process. WebLogic HTTP servlets can access databases, Enterprise JavaBeans, messaging APIs, HTTP sessions, and other facilities of WebLogic Server.

WebLogic Server fully supports HTTP servlets as defined in the Servlet 2.4 specification from Sun Microsystems. HTTP servlets form an integral part of the Java 2 Enterprise Edition (J2EE) standard.

What You Can Do with Servlets

- Create dynamic Web pages that use HTML forms to get end-user input and provide HTML pages that respond to that input. Examples of this utilization include online shopping carts, financial services, and personalized content.
- Create collaborative systems such as online conferencing.
- Have access to a variety of APIs and features by using servlets running in WebLogic Server. For example:
  - Session tracking—Allows a Web site to track a user's progress across multiple Web pages. This functionality supports Web sites such as e-commerce sites that use shopping carts. WebLogic Server supports session persistence to a database, providing fail-over between server down time and session sharing between clustered servers. For more information see "Session Tracking from a Servlet" on page 8-13.
  - JDBC drivers (including BEA)—JDBC drivers provide basic database access. With WebLogic Server's multi-tier JDBC implementations, you can take advantage of connection pools, server-side data caching, and transactions. For more information see "Accessing Databases" on page 7-16.
  - Enterprise JavaBeans—Servlets can use Enterprise JavaBeans (EJB) to encapsulate sessions, data from databases, and other functionality. See "Referencing External EJBs" on page 6-3, "More about the ejb-ref* Elements" on page 6-4, and "Referencing Application-Scoped EJBs" on page 6-4.
  - Java Messaging Service (JMS)—JMS allows your servlets to exchange messages with other servlets and Java programs. See *Programming WebLogic JMS*.
  - Java JDK APIs—Servlets can use the standard Java JDK APIs.
  - Forwarding requests—Servlets can forward a request to another servlet or other resource. "Forwarding a Request" on page 7-18.
- Easily deploy servlets written for any J2EE-compliant servlet engine to WebLogic Server.

Servlet Development Key Points

The following are a few key points relating to servlet development:

- Programmers of HTTP servlets utilize a standard API from JavaSoft, `javax.servlet.http`, to create interactive applications.
- HTTP servlets can read HTTP headers and write HTML coding to deliver a response to a browser client.
- Servlets are deployed to WebLogic Server as part of a Web application. A Web application is a grouping of application components such as servlet classes, JavaServer Pages (JSPs), static HTML pages, images, and security.

Servlets and J2EE

The Servlet 2.4 specification, part of the Java 2 Platform, Enterprise Edition, defines the implementation of the servlet API and the method by which servlets are deployed in enterprise applications. Deploying servlets on a J2EE-compliant server, such as WebLogic Server, is accomplished by packaging the servlets and other resources that make up an enterprise application into a single unit, the Web application. A Web application utilizes a specific directory structure to contain its resources and a deployment descriptor that defines how these resources interact and how the application is accessed by a client. See "The Web Applications Container" on page 2-1.

Java Server Pages

Java Server Pages (JSPs) are Web pages coded with an extended HTML that makes it possible to embed Java code in a Web page. JSPs can call custom Java classes, called *`taglibs`*, using HTML-like tags. The WebLogic `appc` compiler `weblogic.appc` generates JSPs and validates descriptors. You can also precompile JSPs into the `WEB-INF/classes/` directory or as a JAR file under `WEB-INF/lib/` 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.

JSPs are a Sun Microsystems specification for combining Java with HTML to provide dynamic content for Web pages. When you create dynamic content, JSPs are more convenient to write than HTTP servlets because they allow you to embed Java code directly into your HTML pages, in contrast with HTTP servlets, in which you embed HTML inside Java code. JSP is part of the Java 2 Enterprise Edition (J2EE).

JSPs enable you to separate the dynamic content of a Web page from its presentation. It caters to two different types of developers: HTML developers, who are responsible for the graphical design of the page, and Java developers, who handle the development of software to create the dynamic content.

Because JSPs are part of the J2EE standard, you can deploy JSPs on a variety of platforms, including WebLogic Server. In addition, third-party vendors and application developers can provide JavaBean components and define custom JSP tags that can be referenced from a JSP page to provide dynamic content.

What You Can Do with JSPs

- Combine Java with HTML to provide dynamic content for Web pages.
- Call custom Java classes, called *taglibs*, using HTML-like tags.
- Embed Java code directly into your HTML pages, in contrast with HTTP servlets, in which you embed HTML inside Java code.
- Separate the dynamic content of a Web page from its presentation.

Overview of How JSP Requests Are Handled

WebLogic Server handles JSP requests in the following sequence:

1. A browser requests a page with a `.jsp` file extension from WebLogic Server.
2. WebLogic Server reads the request.
3. Using the JSP compiler, WebLogic Server converts the JSP into a servlet class that implements the `javax.servlet.jsp.JspPage` interface. The JSP file is compiled only when the page is first requested, or when the JSP file has been changed. Otherwise, the previously compiled JSP servlet class is re-used, making subsequent responses much quicker.
4. The generated `JspPage` servlet class is invoked to handle the browser request.

It is also possible to invoke the JSP compiler directly without making a request from a browser. For details, see "Using the WebLogic JSP Compiler" on page 10-23.

Because the JSP compiler creates a Java servlet as its first step, you can look at the Java files it produces, or even register the generated `JspPage` servlet class as an HTTP servlet. See "Servlets" on page 2-2.

JSPs and J2EE

BEA WebLogic JSP supports the JSP 2.0 specification from Sun Microsystems. JSP 2.0 includes support for defining custom JSP tag extensions.

Web Application Developer Tools

BEA provides several tools to help you create and configure Web applications. These are discussed in the following sections.

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.

XML Editors

You can use an enterprise-level IDE with DTD validation or another development tool that supports editing of XML files.

Web Application Security

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

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

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

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

P3P Privacy Protocol

The Platform for Privacy Preferences (P3P) provides a way for Web sites to publish their privacy policies in a machine-readable syntax. The WebLogic Server Web application container can support P3P.

There are three ways to tell the browser about the location of the `p3p.xml` file:

- Place the a policy reference file in the "well-known location" (at the location /w3c/p3p.xml on the site).
- Add an extra HTTP header to each response from the Web site giving the location of the policy reference file.
- Place a link to the policy reference file in each HTML page on the site.

For more detailed information, see http://www.w3.org/TR/p3pdeployment#Locating_PRF.

Displaying Special Characters on Linux Browsers

To display special characters on Linux browsers, set the JVM's `file.encoding` system property to `ISO8859_1`. For example, `java -Dfile.encoding=ISO8859_1 weblogic.Server`. For a complete listing, see Sun's "Supported Encodings" page for J2SE 1.4.2.

CHAPTER 3

Creating and Configuring Web Applications

The following sections describe how to create and configure Web application resources.

Directory Structure

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

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

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

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

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

  DefaultWebApp/WEB-INF/web.xml
  : The Web application deployment descriptor that configures the Web application.

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

  DefaultWebApp/WEB-INF/classes
  : Contains server-side classes such as HTTP servlets and utility classes.

  DefaultWebApp/WEB-INF/lib
  : Contains JAR files used by the Web application, including JSP tag libraries.

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

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

Accessing Information in WEB-INF

The WEB-INF directory is not part of the public document tree of the application. No file contained in the WEB-INF directory can be served directly to a client by the container. However,

the contents of the `WEB-INF` directory are visible to servlet code using the `getResource` and `getResourceAsStream()` method calls on the `ServletContext` or includes/forwards using the RequestDispatcher. Hence, if the application developer needs access, from servlet code, to application specific configuration information that should not be exposed directly to the Web client, the application developer may place it under this directory.

Since requests are matched to resource mappings in a case-sensitive manner, client requests for `'/WEB-INF/foo'`, `'/WEb-iNf/foo'`, for example, should not result in contents of the Web application located under `/WEB-INF` being returned, nor any form of directory listing thereof.

Directory Structure Example

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

Listing 3-1 Web Application Directory Structure

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

Main Steps to Create and Configure a Web Application

The following steps summarize the procedure for creating a Web application as part of an Enterprise application using the split development directory structure. See "Creating a Split

Development Directory for an Application," "Building the Applications," and "Deploying the Application" in *Developing Applications with WebLogic Server*.

You may want to use developer tools included with WebLogic Server for creating and configuring Web applications. See "Web Application Developer Tools" on page 2-6.

Step One: Create the Enterprise Application Wrapper

1. Create a directory for your root EAR file:

   `\src\myEAR\`

2. Set your environment as follows:

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

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

3. Package your Enterprise application in the `\src\myEAR\` directory as follows:

   a. Place the Enterprise applications descriptors (`application.xml` and `weblogic-application.xml`) in the `META-INF\` directory. See "Enterprise Application Deployment Descriptors" in *Developing Applications with WebLogic Server*.

   b. Edit the deployment descriptors as needed to fine-tune the behavior of your Enterprise application. See "Web Application Developer Tools" on page 2-6.

   c. Place the Enterprise application `.jar` files in:

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

Step Two: Create the Web Application

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

   `\src\myEAR\myWebApp`

2. Package your Web application in the `\src\myEAR\myWebApp\` directory as follows:

   a. Place the Web application descriptors (`web.xml` and `weblogic.xml`) in the `\src\myEAR\myWebApp\WEB-INF\` directory. See Appendix B, "weblogic.xml Deployment Descriptor Elements."

b. Edit the deployment descriptors as needed to fine-tune the behavior of your Enterprise application. See "Web Application Developer Tools" on page 2-6.

c. Place all HTML files, JSPs, images and any other files referenced by the Web application pages in the root of the Web application:

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

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

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

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

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

Step Three: Creating the build.xml File

Once you have set up your directory structure, you create the `build.xml` file using the `weblogic.BuildXMLGen` utility.

Step Four: Execute the Split Development Directory Structure Ant Tasks

1. Execute the `wlcompile` Ant task to invoke the `javac` compiler. This compiles your Web application Java components into an output directory: `/build/myEAR/WEB-INF/classes`.
2. Execute `wlappc` Ant task to invoke the `appc` compiler. This compiles any JSPs and container-specific EJB classes for deployment.
3. Execute the `wldeploy` Ant task to deploy your Web application as part of an archived or exploded EAR to WebLogic Server.
4. If this is a production environment (rather than development), execute the `wlpackage` Ant task to package your Web application as part of an archived or exploded EAR.

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

Configuring How a Client Accesses a Web Application

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.

Configuring Virtual Hosts for Web Applications

WebLogic Server supports two methods for configuring virtual hosts for Web applications:

- channel based
- host based

Configuring a Channel-based Virtual Host

The following is an example of how to configure a channel-based virtual host:

```
<VirtualHost Name="channel1vh" NetworkAccessPoint="Channel1"
Targets="myserver"/>

<VirtualHost Name="channel2vh" NetworkAccessPoint="Channel2"
Targets="myserver"/>
```

Where `Channel1` and `Channel2` are the names of `NetworkAccessPoint` configured in the `config.xml` file. `NetworkAccessPoint` represents the dedicated server channel name for which the virtual host serves HTTP requests. If the `NetworkAccessPoint` for a given HTTP request does not match the `NetworkAccessPoint` of any virtual host, the incoming HOST header is matched with the `VirtualHostNames` in order to resolve the correct virtual host. If an incoming request does not match a virtual host, the request will be served by the default Web server.

Configuring a Host-based Virtual Host

The following is an example of how to configure a host-based virtual host:

```
<VirtualHost Name="cokevh" Targets="myserver" VirtualHostNames="coke"/>
```

```
<VirtualHost Name="pepsivh" Targets="myserver" VirtualHostNames="pepsi"/>
```

Targeting Web Applications to Virtual Hosts

A Web application component can be targeted to servers and virtual hosts using the WebLogic Administration Console.

If you are migrating from previous versions of WebLogic Server, note that in the config.xml file, all Web application targets must be specified in the targets attribute. The targets attribute has replaced the virtual hosts attribute and a virtual host cannot have the same name as a server or cluster in the same domain. The following is an example of how to target a Web application to a virtual host:

```
<AppDeployment name="test-app" Sourcepath="/myapps/test-app.ear">
   <SubDeployment Name="test-webapp1.war" Targets="virutalhost-1"/>
   <SubDeployment Name="test-webapp2.war" Targets="virtualhost-2"/>
   ...
</AppDeployment>
```

Loading Servlets, Context Listeners, and Filters

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

Order of loading:

1. Context Listeners
2. Filters
3. Servlets

Order of destruction:

1. Servlets
2. Filters
3. Context Listeners

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

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

Shared J2EE Web Application Libraries

A J2EE Web application library is a standalone Web application module registered with the J2EE application container upon deployment. Using WebLogic Server 9.2, multiple Web applications can easily share a single Web application module or collection of modules.

A Web application may reference one or more Web application libraries, but cannot reference other library types (EJBs, EAR files, plain JAR files).

Web application libraries are Web application modules deployed as libraries. They are referenced from the `weblogic.xml` file using the same syntax that is used to reference application libraries in the `weblogic-application.xml` file, except that the `<context-root>` element is ignored.

At deployment time, the classpath of each referenced library is appended to the Web application's classpath. Therefore, the search for all resources and classes occurs first in the original Web application and then in the referenced library.

The deployment tools, appc, wlcompile, and BuildXMLGen support libraries at the Web application level in the same way they support libraries at the application level. For more information about shared J2EE libraries and their deployment, see Creating Shared J2EE Libraries and Optional Packages in *Developing Applications with WebLogic Server*.

Using JSF and JSTL With Web Applications

Three JSF (JavaServer™ Faces) and JSTL (JSP™ Standard Tag Library) packages are bundled with WebLogic Server as WebApp libraries. These libraries can be referenced by standard Web applications that use JSF or JSTL functionality.

For information on referencing these WebApp libraries with your Web applications, see Using WebApp Libraries With Web Applications in *Developing Applications with WebLogic Server*.

The following three packages are being made available as WebApp libraries in release 9.2:

- MyFaces JSF library – http://myfaces.apache.org

- Sun JSF RI library – https://javaserverfaces.dev.java.net
- JSTL library – http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

The libraries are located in the *WL_HOME*/common/deployable-libraries directory. The JSF libraries include the JSTL JAR files for convenience, so that if an application references a JSF library, it automatically gets JSTL support as well.

The following tables list the JSF and JSTL library file names and their MANIFEST entries:

Table 3-1 jsf-myfaces-1.1.1.war

Attribute	Description
Extension-Name	jsf-myfaces
Specification-Title	JavaServer Faces
Specification-Version	1.1
Implementation-Title	MyFaces
Implementation-Version	1.1.1
Implementation-Vendor	MyFaces Project Team.

Table 3-2 jsf-ri-1.1.1.war

Attribute	Description
Extension-Name	jsf-ri
Specification-Title	JavaServer Faces
Specification-Version	1.1
Implementation-Title	'jsf-impl': JavaServer Faces RI
Implementation-Version	1.1.1
Implementation-Vendor	Sun Microsystems, Inc.

Table 3-3 jstl-1.1.2.war

Attribute	Description
`Extension-Name`	jstl
`Specification-Title`	JavaServer Pages Standard Tag Library (JSTL)
`Specification-Version`	1.1
`Implementation-Title`	jakarta-taglibs 'standard': an implementation of JSTL
`Implementation-Version`	1.1.2
`Implementation-Vendor`	Apache Software Foundation

CHAPTER 4

Creating and Configuring Servlets

The following sections describe how to create and configure servlets.

Configuring Servlets

You define servlets as a part of a Web application in several entries in the J2EE standard Web Application deployment descriptor, `web.xml`. The `web.xml` file is located in the `WEB-INF` directory of your Web application.

The first entry, under the root `servlet` element in `web.xml`, defines a name for the servlet and specifies the compiled class that executes the servlet. (Or, instead of specifying a servlet class, you can specify a JSP.) The `servlet` element also contains definitions for initialization attributes and security roles for the servlet.

The second entry in `web.xml`, 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 4-3) 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 2.4 specification.

Listing 4-1 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 4-1 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`
`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.

Table 4-1 url-patterns and Servlet Invocation

URL	Servlet 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`

`ServletServlet can be used` to create a default mappings for servlets. For example, to create a default mapping to map all servlets to `/myservlet/*`, so the servlets can be called using `http://host:port/web-app-name/myservlet/com/foo/FooServlet`, add the following to your `web.xml` file. (The `web.xml` file is located in the `WEB-INF` directory of your Web application.)

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

<servlet-mapping>

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

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 How a Client Accesses a Web Application" on page 3-5.
2. Add a servlet-mapping with url-pattern = "/" as follows:

```
<servlet-mapping>
   <servlet-name>MyOwnDefaultServlet</servlet-name>
   <url-pattern>/myservlet/*(</url-pattern>
</servlet-mapping>
```

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`.

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

Servlet Initialization Attributes

You define initialization attributes for servlets in the Web application deployment descriptor, `web.xml`, in the `init-param` element of the `servlet` element, using `param-name` and `param-value` tags. The `web.xml` file is located in the `WEB-INF` directory of your Web application. For example:

Listing 4-2 Example of Configuring Servlet Initialization Attributes in web.xml

```
<servlet>
  <servlet-name>HelloWorld2</servlet-name>
  <servlet-class>examples.servlets.HelloWorld2</servlet-class>
  <init-param>
    <param-name>greeting</param-name>
    <param-value>Welcome</param-value>
  </init-param>
  <init-param>
    <param-name>person</param-name>
    <param-value>WebLogic Developer</param-value>
  </init-param>
</servlet>
```

Writing a Simple HTTP Servlet

The section provides a procedure for writing a simple HTTP servlet, which prints out the message `Hello World`. A complete code example (the `HelloWorldServlet`) illustrating these steps is included at the end of this section. Additional information about using various J2EE and Weblogic Server services such as JDBC, RMI, and JMS, in your servlet are discussed later in this document.

1. Import the appropriate package and classes, including the following:

   ```
   import javax.servlet.*;
   import javax.servlet.http.*;
   import java.io.*;
   ```

2. Extend `javax.servlet.http.HttpServlet`. For example:

   ```
   public class HelloWorldServlet extends HttpServlet{
   ```

3. Implement a `service()` method.

   The main function of a servlet is to accept an HTTP request from a Web browser, and return an HTTP response. This work is done by the `service()` method of your servlet. Service methods include *response* objects used to create output and *request* objects used to receive data from the client.

You may have seen other servlet examples implement the `doPost()` and/or `doGet()` methods. These methods reply only to POST or GET requests; if you want to handle all request types from a single method, your servlet can simply implement the `service()` method. (However, if you choose to implement the `service()` method, you cannot implement the `doPost()` or `doGet()` methods, unless you call `super.service()` at the beginning of the `service()` method.) The HTTP servlet specification describes other methods used to handle other request types, but all of these methods are collectively referred to as *service* methods.

All the service methods take the same parameter arguments. An `HttpServletRequest` provides information about the request, and your servlet uses an `HttpServletResponse` to reply to the HTTP client. The service method looks like the following:

```
public void service(HttpServletRequest req,
       HttpServletResponse res) throws IOException
{
```

4. Set the content type, as follows:

   ```
   res.setContentType("text/html");
   ```

5. Get a reference to a `java.io.PrintWriter` object to use for output, as follows:

   ```
   PrintWriter out = res.getWriter();
   ```

6. Create some HTML using the `println()` method on the `PrintWriter` object, as shown in the following example:

   ```
   out.println("<html><head><title>Hello World!</title></head>");
   out.println("<body><h1>Hello World!</h1></body></html>");
     }
   }
   ```

7. Compile the servlet, as follows:

   a. Set up a development environment shell with the correct classpath and path settings.

   b. From the directory containing the Java source code for your servlet, compile your servlet into the `WEB-INF/classes` directory of the Web Application that contains your servlet. For example:

      ```
      javac -d /myWebApplication/WEB-INF/classes myServlet.java
      ```

8. Deploy the servlet as part of a Web Application hosted on WebLogic Server.

9. Call the servlet from a browser.

   The URL you use to call a servlet is determined by: (a) the name of the Web Application containing the servlet and (b) the name of the servlet as mapped in the deployment

descriptor of the Web Application. Request parameters can also be included in the URL used to call a servlet.

Generally the URL for a servlet conforms to the following:

`http://host:port/webApplicationName/mappedServletName?parameter`

The components of the URL are defined as follows:

- `host` is the name of the machine running WebLogic Server.
- `port` is the port at which the above machine is listening for HTTP requests.
- `webApplicationName` is the name of the Web Application containing the servlet.
- *parameters* are one or more name-value pairs containing information sent from the browser that can be used in your servlet.

For example, to use a Web browser to call the `HelloWorldServlet` (the example featured in this document), which is deployed in the `examplesWebApp` and served from a WebLogic Server running on your machine, enter the following URL:

```
http://localhost:7001/examplesWebApp/HelloWorldServlet
```

The *host*:*port* portion of the URL can be replaced by a DNS name that is mapped to WebLogic Server.

Advanced Features

The preceding steps create a basic servlet. You will probably also use more advanced features of servlets:

- Handling HTML form data—HTTP servlets can receive and process data received from a browser client in HTML forms.
  - "Retrieving Client Input" on page 7-6
- Application design—HTTP servlets offer many ways to design your application. The following sections provide detailed information about writing servlets:
  - "Providing an HTTP Response" on page 7-4
  - "Threading Issues in HTTP Servlets" on page 7-17
  - "Dispatching Requests to Another Resource" on page 7-17
- Initializing a servlet—if your servlet needs to initialize data, accept initialization arguments, or perform other actions when the servlet is initialized, you can override the `init()` method.

– "Initializing a Servlet" on page 7-2

- Use of *sessions* and *persistence* in your servlet—sessions and persistence allow you to track your users within and between HTTP sessions. Session management includes the use of *cookies*. For more information, see the following sections:
  – "Session Tracking from a Servlet" on page 8-13
  – "Using Cookies in a Servlet" on page 7-11
  – "Configuring Session Persistence" on page 8-4
- Use of WebLogic services in your servlet—WebLogic Server provides a variety of services and APIs that you can use in your Web applications. These services include Java Database Connectivity (JDBC) drivers, JDBC database connection pools, Java Messaging Service (JMS), Enterprise JavaBeans (EJB), and Remote Method Invocation (RMI). For more information, see the following sections:
  – "Using WebLogic Services from an HTTP Servlet" on page 7-15
  – "Accessing Databases" on page 7-16

Complete HelloWorldServlet Example

This section provides the complete Java source code for the example used in the preceding procedure. The example is a simple servlet that provides a response to an HTTP request. Later in this document, this example is expanded to illustrate how to use HTTP parameters, cookies, and session tracking.

Listing 4-3 HelloWorldServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloWorldServlet extends HttpServlet {
  public void service(HttpServletRequest req,
                      HttpServletResponse res)
       throws IOException
  {
    // Must set the content type first
    res.setContentType("text/html");
```

```
    // Now obtain a PrintWriter to insert HTML into
    PrintWriter out = res.getWriter();

    out.println("<html><head><title>" +
                "Hello World!</title></head>");
    out.println("<body><h1>Hello World!</h1></body></html>");
  }
}
```

You can find the source code and instructions for compiling and running examples in the `samples/examples/servlets` directory of your WebLogic Server distribution.

CHAPTER 5

Creating and Configuring JSPs

The following sections describe how to create and configure JSPs.

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 subelements of the `jsp-descriptor` element in 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 subelements, see "jsp-descriptor" on page B-15.

Registering a JSP as a Servlet

You can register a JSP as a servlet using the `servlet` element of the J2EE standard deployment descriptor `web.xml`. (The `web.xml` file is located in the `WEB-INF` directory of your Web application.) A servlet container maintains a map of the servlets known to it. This map is used to resolve requests that are made to the container. Adding entries into this map is known as "registering" a servlet. You add entries to this map by referencing a `servlet` element in `web.xml` through the `servlet-mapping` entry.

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

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

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

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

Configuring JSP Tag Libraries

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

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

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

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

```
<jsp-config>

<taglib>

<taglib-uri>myTaglib</taglib-uri>

<tablig-location>WEB-INF/myTLD.tld</taglib-location>

</taglib>

</jsp-config>
```

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

For more information on creating custom JSP tag libraries, see *Programming JSP Tag Extensions*.

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

- Chapter 13, “Using Custom WebLogic JSP Tags (cache, process, repeat).”
- Chapter 12, “Using WebLogic JSP Form Validation Tags.”

Configuring Welcome Files

Web Application developers can define an ordered list of partial URIs called welcome files in the Web application deployment descriptor. The purpose of this mechanism is to allow the deployer to specify an ordered list of partial URIs for the container to use for appending to URIs when there is a request for a URI that corresponds to a directory entry in the WAR not mapped to a Web component. This feature can make your site easier to use, because the user can type a URL without giving a specific filename.

Note: Welcome files can be JSPs, static pages, or servlets.

Welcome files are defined at the Web application level. If your server is hosting multiple Web applications, you need to define welcome files separately for each Web application. You define Welcome files using the `welcome-file-list` element in `web.xml`. (The `web.xml` file is located in the `WEB-INF` directory of your Web application.) The following is an example Welcome file configuration:

Listing 5-1 Welcome File Example

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

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

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

For more information on this subject, see section 9.10 of the Servlet 2.4 specification.

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 J2EE standard Web application deployment descriptor, `web.xml`. (The `web.xml` file is located in the `WEB-INF` directory of your Web application.)

Determining the Encoding of an HTTP Request

WebLogic Server converts binary (bytes) data contained in an HTTP request to the correct encoding expected by the servlet. The incoming post data might be encoded in a particular encoding that must be converted to the correct encoding on the server side for use in methods such as `request.getParameter(..)`.

There are two ways you can define the code set:

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

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

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

- Because all Web clients do not transmit the information after the semicolon in the above example, you can set the code set to be used for requests by using the `input-charset` element in the WebLogic-specific deployment descriptor, `weblogic.xml`.

  The `java-charset-name` subelement defines the encoding used to convert data when the URL of the request contains the path specified with the `resource-path` subelement.

  This following example ensures that all request parameters that map to the pattern /foo/* are encoded using the Java character set SJIS.

  ```
  <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 on 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. See "charset-mapping" on page B-22.

For example:

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

Configuring Implicit Includes at the Beginning and End of JSPs

You can implicitly include preludes (also called headers) and codas (also called footers) for a group of JSP pages by adding `<include-prelude>` and `<include-coda>` elements respectively within a `<jsp-property-group>` element in the Web application `web.xml` deployment descriptor. Their values are context-relative paths that must correspond to elements in the Web application. When the elements are present, the given paths are automatically included (as in an include directive) at the beginning and end of each JSP page in the property group respectively. When there is more than one include or coda element in a group, they are included in the order they appear. When more than one JSP property group applies to a JSP page, the corresponding elements will be processed in the same order as they appear in the JSP configuration section.

Consider the following files: `/template/prelude.jspf` and `/template/coda.jspf`. These files are used to include code at the beginning and end of each file in the following example:

Listing 5-2 Implicit Includes

```
<jsp-config>
  <jsp-property-group>
    <display-name>WebLogicServer</display-name>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>false</el-ignored>
    <scripting-invalid>false</scripting-invalid>
    <is-xml>false</is-xml>
    <include-prelude>/template/prelude.jspf</include-prelude>
    <include-coda>/template/coda.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

Configuring JSP Property Groups

A JSP property group is a collection of properties that apply to a set of files representing JSP pages. You define these properties in one or more subelements of the `jsp-property-group` element in the `web.xml` deployment descriptor.

Most properties defined in a JSP property group apply to an entire translation unit, that is, the requested JSP file that is matched by its URL pattern and all the files it includes by way of the include directive. The exception is the `page-encoding` property, which applies separately to each JSP file matched by its URL pattern. The applicability of a JSP property group is defined through one or more URL patterns. URL patterns use the same syntax as defined in chapter 11, "Mapping Requests to Servlets" of the Servlet 2.4 specification, but are bound at translation time. All the properties in the property group apply to the resources in the Web application that match any of the URL patterns. There is an implicit property—that of being a JSP file. JSP property groups do not affect tag files.

JSP Property Group Rules

The following are some rules that apply to JSP property groups:

- If a resource matches a URL pattern in both a `servlet-mapping` and a `jsp-property-group`, the pattern that is most specific applies (following the same rules as the servlet specification).
- If the URL patterns are identical, the `jsp-property-group` takes precedence over the `servlet-mapping`.
- If at least one `jsp-property-group` contains the most specific matching URL pattern, the resource is considered to be a JSP file, and the properties in that `jsp-property-group` apply.
- If a resource is considered to be a JSP file, all `include-prelude` and `include-coda` properties apply from all the `jsp-property-group` elements with matching URL patterns. See "Configuring Implicit Includes at the Beginning and End of JSPs" on page 5-6.

What You Can Do with JSP Property Groups

You can configure the `jsp-property-group` to do the following:

- Indicate that a resource is a JSP file (implicit).
- Control disabling of JSP expression language (JSP EL) evaluation.
- Control disabling of Scripting elements.
- Indicate page Encoding information.
- Prelude and Coda automatic includes.
- Indicate that a resource is a JSP document.

For more information on JSP property groups, see chapter 3, "JSP Configuration," of the JSP 2.0 specification.

Writing JSP Documents Using XML Syntax

The JSP 2.0 specification has improved upon the concept of JSP documents by allowing them to leverage XML syntax. Also, JSP documents have been extended to use property groups. A JSP document is a JSP page written using XML syntax. JSP documents need to be described as such, either implicitly or explicitly, to the JSP container, which then processes them as XML documents, checking for well-formedness and applying requests like entity declarations, if present. JSP documents are used to generate dynamic content using the standard JSP semantics.

The following is an example of a simple JSP document that generates, using the JSP standard tag library, an XML document that has `table` as the root element. The table element has three `row` subelements containing values `1`, `2`, and `3`. For more details and other examples, see section 6.4, "Examples of JSP Documents," of the JSP 2.0 specification.

Listing 5-3 Simple JSP Document

```
<table>
<c:forEach
xmlns:c="http://java.sun.com/jsp/jstl/core"
var="counter" begin="1" end="3">
<row>${counter}</row>
</c:forEach>
</table>
```

How to Use JSP Documents

You can use JSP documents in a number of ways including the following:

- JSP documents can be passed directly to the JSP container. This is becoming more important as more and more content is authored in XML. The generated content may be sent directly to a client or it may be part of some XML processing pipeline.
- JSP documents can be manipulated by XML-aware tools.
- JSP documents can be generated from textual representations by applying an XML transformation, such as XSLT.
- A JSP document can be generated automatically, for example, by serializing some objects.

Important Information about JSP Documents

The following are some important pieces of information pertaining to JSP documents:

- By default, files with the filename extension `.jspx` or `.tagx` are treated as JSP documents in the XML syntax.

- JSP property groups defined in the `web.xml` deployment descriptor can control which files in the Web application can be treated as being in the XML syntax. See "Configuring JSP Property Groups" on page 5-7.
- If a JSP file starts with `<jsp:root>`, then it is used in the XML syntax.
- XML namespaces are used instead of `<%@taglib%>` taglib tags (`xmlns:prefix="..."`).
- The `<jsp:scriptlet>`, `<jsp:declaration>` and `<jsp:expression>` tags are used instead of `<%...%>`, `<%!...%>`, and `<%=...%>`.
- The `<jsp:directive.page>` and `<jsp:directive.include>` tags are used instead of `<%@page%>` and `<%@include%>`.
- Inside of attribute values, instead of using `<%=...%>` to denote an expression, only `"%...%"` is used.

For more information on JSP documents, see chapter 6, "JSP Documents," of the JSP 2.0 specification.

CHAPTER 6

Configuring Resources in a Web Application

The following sections describe how to configure Web application resources.

Configuring Resources in a Web Application

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

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. (The `web.xml` file is located in the `WEB-INF` directory of your Web application.) See "Configuring Resources" on page 6-2 for more information.

You can also 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 with scope defined as application are referred to as application-scoped resources. These resources remain private to the application, and application components can access the resource names by adding `<resource-ref>` as explained in "Configuring Resources" on page 6-2.

Configuring Resources

When accessing resources such as a DataSource from a Web application through 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. You can also access JMS module resources, such as queues, topics, and connection factories. For more information see, *Configuring JMS Application Modules for Deployment* in Configuring and Managing WebLogic JMS.

To configure resources:

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

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

Listing 6-1 Using an External DataSource

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

web.xml entries:

<resource-ref>
. . .
```

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

`weblogic.xml` entries:

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

Referencing External EJBs

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

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

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

1. Enter the EJB reference name you use to look up the EJB in your code, the Java class name and the class name of the home and remote interfaces of the EJB in the `ejb-ref` element of the J2EE standard deployment descriptor, `web.xml`. (The `web.xml` file is located in the `WEB-INF` directory of your Web application.)
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 of the J2EE standard deployment descriptor, `web.xml`.

More about the ejb-ref* Elements

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

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

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

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

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

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

  ```
  Context ctx = new InitialContext();

  ctx = (Context)ctx.lookup("java:comp/env");

  Object o = ctx.lookup("ejb1");

  Ejb1Home home = (Ejb1Home) PortableRemoteObject.narrow(o,
  Ejb1Home.class);
  ```

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

Referencing Application-Scoped EJBs

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

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

Listing 6-2 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 J2EE standard 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 6-3 <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` element is referencing should have an entry similar to the following:

Listing 6-4 <ejb-jar> Element

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

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

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

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

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

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

Serving Resources from the CLASSPATH with the ClasspathServlet

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

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

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

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

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

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

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

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

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

Using CGI with WebLogic Server

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

WebLogic Server supports all CGI scripts through an internal WebLogic servlet called the CGIServlet. To use CGI, register the CGIServlet in the Web application deployment descriptor. See "Configuring How a Client Accesses a Web Application" on page 3-5.

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 in the J2EE standard Web application deployment descriptor, web.xml. (The web.xml file is located in the WEB-INF directory of your Web application.) The class name for the CGIServlet is weblogic.servlet.CGIServlet. You do not need to package this class in your Web application.
2. Register the following initialization attributes for the CGIServlet by defining the following init-param elements:

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

`useByteStream`
: By default, character streams are used to read the output of CGI scripts. When scripts produce binary data, the stream may become corrupted due to character encoding. Use the useByteStream parameter to keep the stream from becoming corrupted. Using this parameter for ascii output also improves performance.

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

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

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

Listing 6-5 Example Web Application Deployment Descriptor Entries for Registering the CGIServlet

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

...
```

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

Requesting a CGI Script

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

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

Where

host:port
: Is the host name and port number of WebLogic Server.

myWebApp
: is the name of your Web application.

cgi-bin
: is the `url-pattern` name mapped to the `CGIServlet`.

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

CGI Best Practices

For a list of CGI Best Practices, see "CGI Best Practices" on page C-1.

CHAPTER 7

Servlet Programming Tasks

The following sections describe how to write HTTP servlets in a WebLogic Server environment:

Initializing a Servlet

Normally, WebLogic Server initializes a servlet when the first request is made for the servlet. Subsequently, if the servlet is modified, the `destroy()` method is called on the existing version of the servlet. Then, after a request is made for the modified servlet, the `init()` method of the modified servlet is executed. For more information, see "Servlet Best Practices" on page C-2.

When a servlet is initialized, WebLogic Server executes the `init()` method of the servlet. Once the servlet is initialized, it is not initialized again until you restart WebLogic Server or modify the servlet code. If you choose to override the `init()` method, your servlet can perform certain tasks, such as establishing database connections, when the servlet is initialized. (See "Overriding the init() Method" on page 7-3.)

Initializing a Servlet when WebLogic Server Starts

Rather than having WebLogic Server initialize a servlet when the first request is made for it, you can first configure WebLogic Server to initialize a servlet when the server starts. You do this by specifying the servlet class in the `load-on-startup` element in the J2EE standard Web Application deployment descriptor, `web.xml`. The order in which resources within a Web application are initialized is as follows:

1. ServletContextListeners—the contextCreated() callback for ServletContextListeners registered for this Web application.
2. ServletFilters init() method.
3. Servlet init() method, marked as `load-on-startup` in `web.xml`.

You can pass parameters to an HTTP servlet during initialization by defining these parameters in the Web Application containing the servlet. You can use these parameters to pass values to your servlet every time the servlet is initialized without having to rewrite the servlet.

For example, the following entries in the J2EE standard Web Application deployment descriptor, `web.xml`, define two initialization parameters: `greeting`, which has a value of `Welcome` and `person`, which has a value of `WebLogic Developer`.

```
<servlet>
  ...
 <init-param>
    <description>The salutation</description>
    <param-name>greeting</param-name>
    <param-value>Welcome</param-value>
```

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

To retrieve initialization parameters, call the `getInitParameter(String name)` method from the parent `javax.servlet.GenericServlet` class. When passed the name of the parameter, this method returns the parameter's value as a `String`.

Overriding the init() Method

You can have your servlet execute tasks at initialization time by overriding the `init()` method. The following code fragment reads the `<init-param>` tags that define a greeting and a name in the J2EE standard Web Application deployment descriptor, `web.xml`:

```
String defaultGreeting;
String defaultName;

public void init(ServletConfig config)
    throws ServletException {
  if ((defaultGreeting = getInitParameter("greeting")) == null)
    defaultGreeting = "Hello";

  if ((defaultName = getInitParameter("person")) == null)
    defaultName = "World";
}
```

The values of each parameter are stored in the class instance variables `defaultGreeting` and `defaultName`. The first code tests whether the parameters have null values, and if null values are returned, provides appropriate default values.

You can then use the `service()` method to include these variables in the response. For example:

```
  out.print("<body><h1>");
  out.println(defaultGreeting + " " + defaultName + "!");
  out.println("</h1></body></html>");
```

The `init()` method of a servlet does whatever initialization work is required when WebLogic Server loads the servlet. The default `init()` method does all of the initial work that WebLogic Server requires, so you do not need to override it unless you have special initialization requirements. If you do override `init()`, first call `super.init()` so that the default initialization actions are done first.

Providing an HTTP Response

This section describes how to provide a response to the client in your HTTP servlet. Deliver all responses by using the `HttpServletResponse` object that is passed as a parameter to the `service()` method of your servlet.

1. Configure the `HttpServletResponse`.

   Using the `HttpServletResponse` object, you can set several servlet properties that are translated into HTTP header information:

   - *At a minimum*, set the content type using the `setContentType()` method before you obtain the output stream to which you write the page contents. For HTML pages, set the content type to `text/html`. For example:

     ```
     res.setContentType("text/html");
     ```

   - (optional) You can also use the `setContentType()` method to set the character encoding. For example:

     ```
     res.setContentType("text/html;ISO-88859-4");
     ```

   - Set header attributes using the `setHeader()` method. For dynamic responses, it is useful to set the "`Pragma`" attribute to `no-cache`, which causes the browser to always reload the page and ensures the data is current. For example:

     ```
     res.setHeader("Pragma", "no-cache");
     ```

2. Compose the HTML page.

   The response that your servlet sends back to the client must look like regular HTTP content, essentially formatted as an HTML page.Your servlet returns an HTTP response through an output stream that you obtain using the response parameter of the `service()` method. To send an HTTP response:

   a. Obtain an output stream by using the `HttpServletResponse` object and one of the methods shown in the following two examples:

   - `PrintWriter out = res.getWriter();`
   - `ServletOutputStream out = res.getOutputStream();`

You can use both `PrintWriter` and `ServletOutputStream` in the same servlet (or in another servlet that is included in a servlet). The output of both is written to the same buffer.

b. Write the contents of the response to the output stream using the `print()` method. You can use HTML tags in these statements. For example:

```
out.print("<html><head><title>My Servlet</title>");
out.print("</head><body><h1>");
out.print("Welcome");
out.print("</h1></body></html>");
```

Any time you print data that a user has previously supplied, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. For more information, refer to "Securing Client Input in Servlets" on page 7-10.

Do not close the output stream by using the `close()` method, and avoid flushing the contents of the stream. If you do not close or flush the output stream, WebLogic Server can take advantage of persistent HTTP connections, as described in the next step.

3. Optimize the response.

By default, WebLogic Server attempts to use HTTP persistent connections whenever possible. A persistent connection attempts to reuse the same HTTP TCP/IP connection for a series of communications between client and server. Application performance improves because a new connection need not be opened for each request. Persistent connections are useful for HTML pages containing many in-line images, where each requested image would otherwise require a new TCP/IP connection.

Using the WebLogic Server Administration Console, you can configure the amount of time that WebLogic Server keeps an HTTP connection open.

WebLogic Server must know the length of the HTTP response in order to establish a persistent connection and automatically adds a `Content-Length` property to the HTTP response header. In order to determine the content length, WebLogic Server must buffer the response. However, if your servlet explicitly flushes the `ServletOutputStream`, WebLogic Server cannot determine the length of the response and therefore cannot use persistent connections. For this reason, you should avoid explicitly flushing the HTTP response in your servlets.

You may decide that, in some cases, it is better to flush the response early to display information in the client before the page has completed; for example, to display a banner advertisement while some time-consuming page content is calculated. Conversely, you may want to increase the size of the buffer used by the servlet engine to accommodate a larger response before flushing the response. You can manipulate the size of the response buffer

by using the related methods of the `javax.servlet.ServletResponse` interface. For more information, see the Servlet 2.4 specification.

The default value of the WebLogic Server response buffer is 12K and the buffer size is internally calculated in terms of `CHUNK_SIZE` where `CHUNK_SIZE = 4088 bytes`; if the user sets 5Kb the server rounds the request up to the nearest multiple of `CHUNK_SIZE` which is 2. and the buffer is set to 8176 bytes.

Retrieving Client Input

The HTTP servlet API provides a interface for retrieving user input from Web pages.

An HTTP request from a Web browser can contain more than the URL, such as information about the client, the browser, cookies, and user query parameters. Use query parameters to carry user input from the browser. Use the `GET` method appends parameters to the URL address, and the `POST` method includes them in the HTTP request body.

HTTP servlets need not deal with these details; information in a request is available through the `HttpServletRequest` object and can be accessed using the `request.getParameter()` method, regardless of the send method.

Read the following for more detailed information about the ways to send query parameters from the client:

- Encode the parameters directly into the URL of a link on a page. This approach uses the `GET` method for sending parameters. The parameters are appended to the URL after a `?` character. Multiple parameters are separated by a `&` character. Parameters are always specified in *name=value* pairs so the order in which they are listed is not important. For example, you might include the following link in a Web page, which sends the parameter `color` with the value `purple` to an HTTP servlet called `ColorServlet`:

  ```
  <a href=
     "http://localhost:7001/myWebApp/ColorServlet?color=purple">
     Click Here For Purple!</a>
  ```

- Manually enter the URL, with query parameters, into the browser location field. This is equivalent to clicking the link shown in the previous example.

- Query the user for input with an HTML form. The contents of each user input field on the form are sent as query parameters when the user clicks the form's Submit button. Specify the method used by the form to send the query parameters (`POST` or `GET`) in the `<FORM>` tag using the `METHOD="GET|POST"` attribute.

Query parameters are always sent in *name=value* pairs, and are accessed through the `HttpServletRequest` object. You can obtain an `Enumeration` of all parameter names in a query, and fetch each parameter value by using its parameter name. A parameter usually has only one value, but it can also hold an array of values. Parameter values are always interpreted as `Strings`, so you may need to cast them to a more appropriate type.

The following sample from a `service()` method examines query parameter names and their values from a form. Note that `request` is the `HttpServletRequest` object.

```
Enumeration params = request.getParameterNames();
String paramName = null;
String[] paramValues = null;

while (params.hasMoreElements()) {
  paramName = (String) params.nextElement();
  paramValues = request.getParameterValues(paramName);
  System.out.println("\nParameter name is " + paramName);
  for (int i = 0; i < paramValues.length; i++) {
    System.out.println(", value " + i + " is " +
                       paramValues[i].toString());
  }
}
```

Note: Any time you print data that a user has supplied, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. For more information, refer to "Securing Client Input in Servlets" on page 7-10.

Methods for Using the HTTP Request

This section defines the methods of the `javax.servlet.HttpServletRequest` interface that you can use to get data from the request object. You should keep the following limitations in mind:

- You cannot read request parameters using any of the `getParameter()` methods described in this section and then attempt to read the request with the `getInputStream()` method.
- You cannot read the request with `getInputStream()` and then attempt to read request parameters with one of the `getParameter()` methods.

If you attempt either of the preceding procedures, an `illegalStateException` is thrown.

You can use the following methods of `javax.servlet.HttpServeletRequest` to retrieve data from the request object:

`HttpServletRequest.getMethod()`
: Allows you to determine the request method, such as `GET` or `POST`.

`HttpServletRequest.getQueryString()`
: Allows you to access the query string. (The remainder of the requested URL, following the `?` character.)

`HttpServletRequest.getParameter()`
: Returns the value of a parameter.

`HttpServletRequest.getParameterNames()`
: Returns an array of parameter names.

`HttpServletRequest.getParameterValues()`
: Returns an array of values for a parameter.

`HttpServletRequest.getInputStream()`
: Reads the body of the request as binary data. If you call this method after reading the request parameters with `getParameter()`, `getParameterNames()`, or `getParameterValues()`, an `illegalStateException` is thrown.

Example: Retrieving Input by Using Query Parameters

In this example, the `HelloWorld2.java` servlet example is modified to accept a username as a query parameter, in order to display a more personal greeting. The `service()` method is shown here.

Listing 7-1 Retrieving Input with the `service()` Method

```
public void service(HttpServletRequest req,
         HttpServletResponse res)
     throws IOException
{
  String name, paramName[];
  if ((paramName = req.getParameterValues("name"))
      != null) {
    name = paramName[0];
  }
  else {
```

```
    name = defaultName;
  }

  // Set the content type first
  res.setContentType("text/html");
  // Obtain a PrintWriter as an output stream
  PrintWriter out = res.getWriter();

  out.print("<html><head><title>" +
              "Hello World!" + </title></head>");
  out.print("<body><h1>");
  out.print(defaultGreeting + " " + name + "!");
  out.print("</h1></body></html>");
}
```

The `getParameterValues()` method retrieves the value of the `name` parameter from the HTTP query parameters. You retrieve these values in an array of type `String`. A single value for this parameter is returned and is assigned to the first element in the `name` array. If the parameter is not present in the query data, `null` is returned; in this case, `name` is assigned to the default name that was read from the `<init-param>` by the `init()` method.

Do not base your servlet code on the assumption that parameters are included in an HTTP request. The `getParameter()` method has been deprecated; as a result, you might be tempted to shorthand the `getParameterValues()` method by tagging an array subscript to the end. However, this method can return `null` if the specified parameter is not available, resulting in a `NullPointerException`.

For example, the following code triggers a `NullPointerException`:

```
String myStr = req.getParameterValues("paramName")[0];
```

Instead, use the following code:

```
if ((String myStr[] =
          req.getParameterValues("paramName"))!=null) {
  // Now you can use the myStr[0];
}
else {
```

```
  // paramName was not in the query parameters!
}
```

Securing Client Input in Servlets

The ability to retrieve and return user-supplied data can present a security vulnerability called ***cross-site scripting***, which can be exploited to steal a user's security authorization. For a detailed description of cross-site scripting, refer to "Understanding Malicious Content Mitigation for Web Developers" (a CERT security advisory) at http://www.cert.org/tech_tips/malicious_code_mitigation.html.

To remove the security vulnerability, before you return data that a user has supplied, scan the data for any of the HTML special characters in Table 7-1. If you find any special characters, replace them with their HTML entity or character reference. Replacing the characters prevents the browser from executing the user-supplied data as HTML.

Table 7-1 HTML Special Characters that Must Be Replaced

Replace this special character:	With this entity/character reference:
<	<
>	>
(	&40;
)	&41;
#	&35;
&	&38;

Using a WebLogic Server Utility Method

WebLogic Server provides the `weblogic.servlet.security.Utils.encodeXSS()` method to replace the special characters in user-supplied data. To use this method, provide the user-supplied data as input. For example, to secure the user-supplied data in Listing 7-1, replace the following line:

```
out.print(defaultGreeting + " " + name + "!");
```

with the following:

```
out.print(defaultGreeting + " " +
weblogic.security.servlet.encodeXSS(name) + "!");
```

To secure an entire application, you must use the `encodeXSS()` method ***each time*** you return user-supplied data. While the previous example in Listing 7-1 is an obvious location in which to use the `encodeXSS()` method, Table 7-2 describes other locations to consider.

Table 7-2 Code that Returns User-Supplied Data

Page Type	User-Supplied Data	Example
Error page	Erroneous input string, invalid URL, username	An error page that says "*`username`* is not permitted access."
Status page	Username, summary of input from previous pages	A summary page that asks a user to confirm input from previous pages.
Database display	Data presented from a database	A page that displays a list of database entries that have been previously entered by a user.

Using Cookies in a Servlet

A cookie is a piece of information that the server asks the client browser to save locally on the user's disk. Each time the browser visits the same server, it sends all cookies relevant to that server with the HTTP request. Cookies are useful for identifying clients as they return to the server.

Each cookie has a name and a value. A browser that supports cookies generally allows each server domain to store up to 20 cookies of up to 4k per cookie.

Setting Cookies in an HTTP Servlet

To set a cookie on a browser, create the cookie, give it a value, and add it to the `HttpServletResponse` object that is the second parameter in your servlet's service method. For example:

```
Cookie myCookie = new Cookie("ChocolateChip", "100");
myCookie.setMaxAge(Integer.MAX_VALUE);
response.addCookie(myCookie);
```

This examples shows how to add a cookie called `ChocolateChip` with a value of `100` to the browser client when the response is sent. The expiration of the cookie is set to the largest possible value, which effectively makes the cookie last forever. Because cookies accept only string-type values, you should cast to and from the desired type that you want to store in the cookie. When using EJBs, a common practice is to use the *home handle* of an EJB instance for the cookie value and to store the user's details in the EJB for later reference.

Retrieving Cookies in an HTTP Servlet

You can retrieve a cookie object from the `HttpServletRequest` that is passed to your servlet as an argument to the `service()` method. The cookie itself is presented as a `javax.servlet.http.Cookie` object.

In your servlet code, you can retrieve all the cookies sent from the browser by calling the `getCookies()` method. For example:

```
Cookie[] cookies = request.getCookies();
```

This method returns an array of all cookies sent from the browser, or `null` if no cookies were sent by the browser. Your servlet must process the array in order to find the correct named cookie. You can get the name of a cookie using the `Cookie.getName()` method. It is possible to have more that one cookie with the same name, but different path attributes. If your servlets set multiple cookies with the same names, but different path attributes, you also need to compare the cookies by using the `Cookie.getPath()` method. The following code illustrates how to access the details of a cookie sent from the browser. It assumes that all cookies sent to this server have unique names, and that you are looking for a cookie called `ChocolateChip` that may have been set previously in a browser client.

```
Cookie[] cookies = request.getCookies();
boolean cookieFound = false;

for(int i=0; i < cookies.length; i++) {
  thisCookie = cookies[i];
  if (thisCookie.getName().equals("ChocolateChip")) {
    cookieFound = true;
    break;
  }
}

if (cookieFound) {
  // We found the cookie! Now get its value
```

```
  int cookieOrder = String.parseInt(thisCookie.getValue());
}
```

Using Cookies That Are Transmitted by Both HTTP and HTTPS

Because HTTP and HTTPS requests are sent to different ports, some browsers may not include the cookie sent in an HTTP request with a subsequent HTTPS request (or vice-versa). This may cause new sessions to be created when servlet requests alternate between HTTP and HTTPS. To ensure that all cookies set by a specific domain are sent to the server every time a request in a session is made, set the `cookie-domain` element to the name of the domain. The `cookie-domain` element is a sub-element of the `session-descriptor` element in the WebLogic-specific deployment descriptor `weblogic.xml`. For example:

```
<session-descriptor>
  <cookie-domain>mydomain.com</cookie-domain>
</session-descriptor>
```

The `cookie-domain` element instructs the browser to include the proper cookie(s) for all requests to hosts in the domain specified by `mydomain.com`. For more information about this property or configuring session cookies, see "Setting Up Session Management" on page 8-1.

Application Security and Cookies

Using cookies that enable automatic account access on a machine is convenient, but can be undesirable from a security perspective. When designing an application that uses cookies, follow these guidelines:

- Do not assume that a cookie is always correct for a user. Sometimes machines are shared or the same user may want to access a different account.
- Allow your users to make a choice about leaving cookies on the server. On shared machines, users may not want to leave automatic logins for their account. Do not assume that users know what a cookie is; instead, ask a question like:

  `Automatically login from this computer?`
- Always ask for passwords from users logging on to obtain sensitive data. Unless a user requests otherwise, you can store this preference and the password in the user's session data. Configure the session cookie to expire when the user quits the browser.

Response Caching

The cache filter works similarly to the cache tag with the following exceptions:

- It caches on a page level (or included page) instead of a JSP fragment level.
- Instead of declaring the caching parameters inside the document you can declare the parameters in the configuration of the Web application.

The cache filter has some default behavior that the cache tag does not for pages that were not included from another page. The cache filter automatically caches the response headers Content-Type and Last-Modified. When it receives a request that results in a cached page it compares the If-Modified-Since request header to the Last-Modified response header to determine whether it needs to actually serve the content or if it can send an `302 SC_NOT_MODIFED` status with an empty content instead.

The following example shows how to register a cache filter to cache all the HTML pages in a Web application using the `filter` element of the J2EE standard deployment descriptor, `web.xml`.

```
<filter>
  <filter-name>HTML</filter-name>
  <filter-class>weblogic.cache.filter.CacheFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>HTML</filter-name>
  <url-pattern>*.html</url-pattern>
</filter-mapping>
```

The cache system uses soft references for storing the cache. So the garbage collector might or might not reclaim the cache depending on how recently the cache was created or accessed. It will clear the soft references in order to avoid throwing an OutOfMemoryError.

Initialization Parameters

To make sure that if the web pages were updated at some point you got the new copies into the cache, you could add a timeout to the filter. Using the init-params you can set many of the same parameters that you can set for the cache tag:

The initialization parameters are

- `Name` This is the name of the cache. It defaults to the request URI for compatibility with *.extension URL patterns.

- `Timeout` This is the amount of time since the last cache update that the filter waits until trying to update the content in the cache again. The default unit is seconds but you can also specify it in units of ms (milliseconds), s (seconds), m (minutes), h (hours), or d (days).
- `Scope` The scope of the cache can be any one of *request*, *session*, *application*, or *cluster*. Request scope is sometimes useful for looping constructs in the page and not much else. The scope defaults to *application*. To use *cluster* scope you must set up the *ClusterListener*.
- `Key` This specifies that the cache is further specified not only by the *name* but also by values of various entries in scopes. These are specified just like the keys in the *CacheTag* although you do not have *page* scope available.
- `Vars` These are the variables calculated by the page that you want to cache. Typically this is used with servlets that pull information out of the database based on input parameters.
- `Size` This limits the number of different unique key values cached. It defaults to infinity.

The following example shows where the `init-parameter` is located in the filter code.

```
<filter>
  <filter-name>HTML</filter-name>
  <filter-class>weblogic.cache.filter.CacheFilter</filter-class>
  <init-param>
```

- `Max-cache-size` This limits the `size of an element added to the cache. It defaults to 64k.`

Using WebLogic Services from an HTTP Servlet

When you write an HTTP servlet, you have access to many rich features of WebLogic Server, such as JNDI, EJB, JDBC, and JMS.

The following documents provide additional information about these features:

- *Programming WebLogic EJB*
- *Programming WebLogic JDBC*
- *Programming WebLogic JNDI*
- *Programming WebLogic JMS*

Accessing Databases

WebLogic Server supports the use of Java Database Connectivity (JDBC) from server-side Java classes, including servlets. JDBC allows you to execute SQL queries from a Java class and to process the results of those queries. For more information on JDBC and WebLogic Server, see *Using WebLogic JDBC*.

You can use JDBC in servlets as described in the following sections:

- "Connecting to a Database Using a DataSource Object" on page 7-16.
- "Connecting Directly to a Database Using a JDBC Driver" on page 7-17.

Connecting to a Database Using a DataSource Object

A `DataSource` is a server-side object that references a connection pool. The connection pool registration defines the JDBC driver, database, login, and other parameters associated with a database connection. You create DataSource objects and connection pools through the Administration Console. *Using a* `DataSource` *object is recommended when creating J2EE-compliant applications*.

Using a DataSource in a Servlet

1. Register a connection pool using the Administration Console. For more information, see "JDBC Data Source: Configuration: Connection Pool".
2. Register a `DataSource` object that points to the connection pool.
3. Look up the `DataSource` object in the JNDI tree. For example:

```
Context ctx = null;

// Get a context for the JNDI look up
ctx = new InitialContext(ht);

// Look up the DataSource object
javax.sql.DataSource ds
        = (javax.sql.DataSource) ctx.lookup ("myDataSource");
```

4. Use the `DataSource` to create a JDBC connection. For example:

```
java.sql.Connection conn = ds.getConnection();
```

5. Use the connection to execute SQL statements. For example:

```
Statement stmt = conn.createStatement();
stmt.execute("select * from emp");
```

. . .

Connecting Directly to a Database Using a JDBC Driver

Connecting directly to a database is the least efficient way of making a database connection because a new database connection must be established for each request. You can use any JDBC driver to connect to your database. BEA provides JDBC drivers for Oracle and Microsoft SQL Server. For more information, see *Programming WebLogic JDBC*.

Threading Issues in HTTP Servlets

When you design a servlet, you should consider how the servlet is invoked by WebLogic Server under high load. It is inevitable that more than one client will hit your servlet simultaneously. Therefore, write your servlet code to guard against sharing violations on shared resources or instance variables.

It is recommended that shared-resource issues be handled on an individual servlet basis. Consider the following guidelines:

- Wherever possible, avoid synchronization, because it causes subsequent servlet requests to bottleneck until the current thread completes.
- Define variables that are specific to each servlet request within the scope of the service methods. Local scope variables are stored on the stack and, therefore, are not shared by multiple threads running within the same method, which avoids the need to be synchronized.
- Access to external resources should be synchronized on a Class level, or encapsulated in a transaction.

Dispatching Requests to Another Resource

This section provides an overview of commonly used methods for dispatching requests from a servlet to another resource.

A servlet can pass on a request to another resource, such as a servlet, JSP, or HTML page. This process is referred to as *request dispatching*. When you dispatch requests, you use either the `include()` or `forward()` method of the `RequestDispatcher` interface.

For a complete discussion of request dispatching, see section 8.2 of the Servlet 2.4 specification (see `http://java.sun.com/products/servlet/download.html#specs`) from Sun Microsystems.

By using the `RequestDispatcher`, you can avoid sending an HTTP-redirect response back to the client. The `RequestDispatcher` passes the HTTP request to the requested resource.

To dispatch a request to a particular resource:

1. Get a reference to a `ServletContext`:

   ```
   ServletContext sc = getServletConfig().getServletContext();
   ```

2. Look up the `RequestDispatcher` object using one of the following methods:

   - `RequestDispatcher rd = sc.getRequestDispatcher(String` *`path`*);

     `path` should be relative to the root of the Web Application.

   - `RequestDispatcher rd = sc.getNamedDispatcher(String` *`name`*);

     Replace *`name`* with the name assigned to the servlet in the J2EE standard Web Application deployment descriptor, `web.xml`, with the `<servlet-name>` element.

   - `RequestDispatcher rd = ServletRequest.getRequestDispatcher(String path);`

     This method returns a `RequestDispatcher` object and is similar to the `ServletContext.getRequestDispatcher(String` *`path`*) method except that it allows the *`path`* specified to be relative to the current servlet. If the path begins with a / character it is interpreted to be relative to the Web Application.

   You can obtain a `RequestDispatcher` for any HTTP resource within a Web Application, including HTTP Servlets, JSP pages, or plain HTML pages by requesting the appropriate URL for the resource in the `getRequestDispatcher()` method. Use the returned `RequestDispatcher` object to forward the request to another servlet.

3. Forward or include the request using the appropriate method:

   - `rd.forward(request,response);`
   - `rd.include(request,response);`

   These methods are discussed in the next two sections.

Forwarding a Request

Once you have the correct `RequestDispatcher`, your servlet forwards a request using the `RequestDispatcher.forward()` method, passing `HTTPServletRequest` and `HTTPServletResponse` as arguments. If you call this method when output has already been sent to the client an `IllegalStateException` is thrown. If the response buffer contains pending output that has not been committed, the buffer is reset.

The servlet must not attempt to write any previous output to the response. If the servlet retrieves the `ServletOutputStream` or the `PrintWriter` for the response before forwarding the request, an `IllegalStateException` is thrown.

All other output from the original servlet is ignored after the request has been forwarded.

If you are using any type of authentication, a forwarded request, by default, does not require the user to be re-authenticated. You can change this behavior to require authentication of a forwarded request by adding the `check-auth-on-forward/` element to the `container-descriptor` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. For example:

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

Including a Request

Your servlet can include the output from another resource by using the `RequestDispatcher.include()` method, and passing `HTTPServletRequest` and `HTTPServletResponse` as arguments. When you include output from another resource, the included resource has access to the request object.

The included resource can write data back to the `ServletOutputStream` or `Writer` objects of the response object and then can either add data to the response buffer or call the `flush()` method on the response object. Any attempt to set the response status code or to set any HTTP header information from the included servlet response is ignored.

In effect, you can use the `include()` method to mimic a “server-side-include” of another HTTP resource from your servlet code.

RequestDispatcher and Filters

The Servlet 2.3 Specification from Sun Microsystems did not specify whether filters should be applied on forwards and includes. The Servlet 2.4 specification clarifies this by introducing a new `dispatcher` element in the `web.xml` deployment descriptor. Using this `dispatcher` element, you can configure a `filter-mapping` to be applied on `REQUEST`/`FORWARD`/`INCLUDE`/`ERROR`. In WebLogic Server 8.1, the default was `REQUEST+FORWARD+INCLUDE`. For the old DTD-based deployment descriptors, the default value has not been changed in order to preserve backward compatibility. For the new descriptors (schema based) the default is `REQUEST`.

You can change the default behavior of dispatched requests by setting the `filter-dispatched-requests-enabled` element in `weblogic.xml`. This element controls

whether or not filters are applied to dispatched (include/forward) requests. The default value for old DTD-based deployment descriptors is `true`. The default for the new schema-based descriptors is `false`

For more information about RequestDispatcher and filters, see section 6.2.5 of the Servlet 2.4 specification. For more information about writing and configuring filters for WebLogic Server, see Chapter 11, "Filters."

Proxying Requests to Another Web Server

The following sections discuss how to proxy HTTP requests to another Web server:

Overview of Proxying Requests to Another Web Server

When you use WebLogic Server as your primary Web server, you may also want to configure WebLogic Server to pass on, or proxy, certain requests to a secondary Web server, such as Netscape Enterprise Server, Apache, or Microsoft Internet Information Server. Any request that gets proxied is redirected to a specific URL.You can even proxy to another Web server on a different machine.You proxy requests based on the URL of the incoming request.

The `HttpProxyServlet` (provided as part of the distribution) takes an HTTP request, redirects it to the proxy URL, and sends the response to the client's browser back through WebLogic Server. To use the `HttpProxyServlet`, you must configure it in a Web Application and deploy that Web Application on the WebLogic Server that is redirecting requests.

Setting Up a Proxy to a Secondary Web Server

To set up a proxy to a secondary HTTP server:

1. Register the `proxy` servlet in your Web Application deployment descriptor (see "Sample web.xml for Use with ProxyServlet" on page 7-22). The Web Application must be the default Web Application of the server instance that is responding to requests. The class name for the proxy servlet is `weblogic.servlet.proxy.HttpProxyServlet`.

2. Define an initialization parameter for the `ProxyServlet` with a `<param-name>` of `redirectURL` and a `<param-value>` containing the URL of the server to which proxied requests should be directed.
3. Optionally, define the following `<KeyStore>` initialization parameters to use two-way SSL with your own identity certificate and key. If no `<KeyStore>` is specified in the deployment descriptor, the proxy will assume one-way SSL.
   - `<KeyStore>` – The key store location in your Web application.
   - `<KeyStoreType>` – The key store type. If it is not defined, the default type will be used instead.
   - `<PrivateKeyAlias>` – The private key alias.
   - `<KeyStorePasswordProperties>` – A property file in your Web application that defines encrypted passwords to access the key store and private key alias. The file contents looks like this:

     ```
     KeyStorePassword={3DES}i4+50LCKenQO8BBvlsXTrg\=\=
     PrivateKeyPassword={3DES}a4TcG4mtVVBRKtZwH3p7yA\=\=
     ```

     You must use the `weblogic.security.Encrypt` command-line utility to encrypt the password. For more information on the `Encrypt` utility, as well as the `CertGen`, and `der2pem` utilities, see Using the WebLogic Server Java Utilities in the *WebLogic Server Command Reference*.
4. Map the `ProxyServlet` to a `<url-pattern>`. Specifically, map the file extensions you wish to proxy, for example `*.jsp`, or `*.html`. Use the `<servlet-mapping>` element in the `web.xml` Web Application deployment descriptor.

   If you set the `<url-pattern>` to "`/`", then any request that cannot be resolved by WebLogic Server is proxied to the remote server. However, you must also specifically map the following extensions: `*.jsp`, `*.html`, and `*.html` if you want to proxy files ending with those extensions.
5. Deploy the Web Application on the WebLogic Server instance that redirects incoming requests.

Sample Deployment Descriptor for the Proxy Servlet

The following is an sample of a Web applications deployment descriptor for using the `ProxyServlet`.

Listing 7-2 Sample web.xml for Use with ProxyServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://java.sun.com/xml/ns/j2ee"
            xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="2.4">
<web-app>
<servlet>
  <servlet-name>ProxyServlet</servlet-name>
  <servlet-class>weblogic.servlet.proxy.HttpProxyServlet</servlet-class>
  <init-param>
    <param-name>redirectURL</param-name>
    <param-value>http://server:port</param-value>
  </init-param>
  <init-param>
    <param-name>KeyStore</param-name>
    <param-value>/mykeystore</param-value>
  </init-param>
  <init-param>
    <param-name>KeyStoreType</param-name>
    <param-value>jks</param-value>
  </init-param>
  <init-param>
    <param-name>PrivateKeyAlias</param-name>
    <param-value>passalias</param-value>
  </init-param>
  <init-param>
    <param-name>KeyStorePasswordProperties</param-name>
    <param-value>mykeystore.properties</param-value>
  </init-param>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>ProxyServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ProxyServlet</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ProxyServlet</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ProxyServlet</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>

</web-app>
```

Clustering Servlets

Clustering servlets provides failover and load balancing benefits. To deploy a servlet in a WebLogic Server cluster, deploy the Web Application containing the servlet on all servers in the cluster.

For information on requirements for clustering servlets, and to understand the connection and failover processes for requests that are routed to clustered servlets, see "Replication and Failover for Servlets and JSPs" in *Using WebLogic Server Clusters*.

Note: Automatic failover for servlets requires that the servlet session state be replicated in memory. For instructions, see "Configure In-Memory HTTP Replication" in *Using WebLogic Server Clusters*.

For information on the load balancing support that a WebLogic Server cluster provides for servlets, and for related planning and configuration considerations for architects and administrators, see "Load Balancing for Servlets and JSPs" in *Using WebLogic Server Clusters*.

Referencing a Servlet in a Web Application

The URL used to reference a servlet in a Web Application is constructed as follows:

```
http://myHostName:port/myContextPath/myRequest/?myRequestParameters
```

The components of this URL are defined as follows:

`myHostName`
: The DNS name mapped to the Web Server defined in the WebLogic Server Administration Console.

  This portion of the URL can be replaced with `host:port`, where `host` is the name of the machine running WebLogic Server and `port` is the port at which WebLogic Server is listening for requests.

`port`
: The port at which WebLogic Server is listening for requests. The Servlet can communicate with the proxy only through the listenPort on the Server mBean and the SSL mBean.

`myContextPath`
: The name of the context root which is specified in the `weblogic.xml file`, or the uri of the web module which is specified in the `config.xml file`.

`myRequest`
: The name of the servlet as defined in the `web.xml` file.

`myRequestParameters`
: Optional HTTP request parameters encoded in the URL, which can be read by an HTTP servlet.

URL Pattern Matching

WebLogic Server provides the user with the ability to implement a URL matching utility which does not conform to the J2EE rules for matching. The utility must be configured in the `weblogic.xml` deployment descriptor rather than the web.xml deployment descriptor used for the configuration of the default implementation of `URLMatchMap`.

To be used with WebLogic Server, the URL matching utility must implement the following interface:

```
Package weblogic.servlet.utils;
public interface URLMapping {
  public void put(String pattern, Object value);
```

```
  public Object get(String uri);
  public void remove(String pattern)
  public void setDefault(Object defaultObject);
  public Object getDefault();
  public void setCaseInsensitive(boolean ci);
  public boolean isCaseInsensitive();
  public int size();
  public Object[] values();
  public String[] keys();
}
```

The SimpleApacheURLMatchMap Utility

The included `SimpleApacheURLMatchMap` utility is not J2EE specific. It can be configured in the `weblogic.xml` deployment descriptor file and allows the user to specify Apache style pattern matching rather than the default URL pattern matching provided in the `web.xml` deployment descriptor. For more information, see "url-match-map" on page B-24.

A Future Response Model for HTTP Servlets

In general, WebLogic Server processes incoming HTTP requests and the response is returned immediately to the client. Such connections are handled synchronously by the same thread. However, some HTTP requests may require longer processing time. Database connection, for example, may create longer response times. Handling these requests synchronously causes the thread to be held, waiting until the request is processed and the response sent.

To avoid this hung-thread scenario, WebLogic Server provides two classes that handle HTTP requests asynchronously by de-coupling the response from the thread that handles the incoming request. The following sections describe these classes.

Abstract Asynchronous Servlet

The Abstract Asynchronous Servlet enables you to handle incoming requests and servlet responses with different threads. This class explicitly provides a better general framework for handling the response than the Future Response Servlet, including thread handling.

You implement the Abstract Asynchronous Servlet by extending the `weblogic.servlet.http.AbstractAsyncServlet.java` class. This class provides the following abstract methods that you must override in your extended class.

doRequest

This method processes the servlet request. The following code example demonstrates how to override this method.

Listing 7-3 Overriding doRequest in AbstractAsynchServlet.java

```
public boolean doRequest(RequestResponseKey rrk)
      throws ServletException, IOException {
      HttpServletRequest req = rrk.getRequest();
      HttpServletResponse res = rrk.getResponse();

      if (req.getParameter("immediate") != null) {
            res.setContentType("text/html");
            PrintWriter out = res.getWriter();
            out.println("Hello World Immediately!");
            return false ;
      }
      else {
            TimerManagerFactory.getTimerManagerFactory()
            .getDefaultTimerManager().schedule
            (new TimerListener() {
                  public void timerExpired(Timer timer)
                        {try {
                              AbstractAsyncServlet.notify(rrk, null);
                        }
                        catch (Exception e) {
                              e.printStackTrace();
                        }
                  }
            }, 2000);
      return true;
      }
}
```

doResponse

This method processes the servlet response.

Note: The servlet instance that processed the **doRequest()** method used to handle the original incoming request method will not necessarily be the one to process the `doResponse()` method.

If an exception occurs during processing, the container returns an error to the client. The following code example demonstrates how to override this method.

Listing 7-4 Overriding doResponse() in AbstractAsyncServlet.java

```
public void doResponse (RequestResponseKey rrk, Object context)
   throws ServletException, IOException
      {
      HttpServletRequest req = rrk.getRequest();
      HttpServletResponse res = rrk.getResponse();

      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      out.println("Hello World!");
}
```

doTimeOut

This method sends a servlet response error when the `notify()` method is not called within the timeout period.

Note: The servlet instance that processed the doRequest() method used to handle the original incoming request method will not necessarily be the one to process the doTimeOut() method.

Listing 7-5 Overriding doTimeOut() in AbstractAsyncServlet.java

```
public void doTimeout (RequestResponseKey rrk)
      throws ServletException, IOException
{
      HttpServletRequest req = rrk.getRequest();
      HttpServletResponse res = rrk.getResponse();

      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
```

```
      out.println("Timeout!");
}
```

Future Response Servlet

Although BEA recommends using the Abstract Asynchronous Servlet, you can also use the Future Response Servlet to handle servlet responses with a different thread than the one that handles the incoming request. You enable this servlet by extending `weblogic.servlet.FutureResponseServlet.java`, which gives you full control over how the response is handled and allows more control over thread handling. However, using this class to avoid hung threads requires you to provide most of the code.

The exact implementation depends on your needs, but you must override the `service()` method of this class at a minimum. The following example shows how you can override the service method.

Listing 7-6 Overriding the service() method of FutureResponseServlet.java

```
public void service(HttpServletRequest req, FutureServletResponse rsp)
  throws IOException, ServletException {
  if(req.getParameter("immediate") != null){
    PrintWriter out = rsp.getWriter();
    out.println("Immediate response!");
    rsp.send();
  } else {
    Timer myTimer = new Timer();
    MyTimerTask mt = new MyTimerTask(rsp, myTimer);
    myTimer.schedule(mt, 100);
  }
}

private static class MyTimerTask extends TimerTask{
  private FutureServletResponse rsp;
  Timer timer;
  MyTimerTask(FutureServletResponse rsp, Timer timer){
    this.rsp = rsp;
    this.timer = timer;
  }
```

```
  public void run(){
    try{
      PrintWriter out = rsp.getWriter();
      out.println("Delayed Response");
      rsp.send();
      timer.cancel();
    }
    catch(IOException e){
      e.printStackTrace();
    }
  }
}
```

CHAPTER 8

Using Sessions and Session Persistence

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

Overview of HTTP Sessions

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

Setting Up Session Management

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

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

You can also store data permanently from an HTTP session. See "Configuring Session Persistence" on page 8-4.

HTTP Session Properties

You configure WebLogic Server session tracking by defining properties in the WebLogic-specific deployment descriptor, `weblogic.xml`. For a complete list of session attributes, see "session-descriptor" on page B-7.

In a previous WebLogic Server release, a change was introduced to the SessionID format that caused some load balancers to lose the ability to retain session stickiness. A server startup flag, `-Dweblogic.servlet.useExtendedSessionFormat=true`, retains the information that the load-balancing application needs for session stickiness. The extended session ID format will be part of the URL if URL rewriting is activated, and the startup flag is set to true.

Session Timeout

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

- Set the `timeout-secs` parameter value in the `session-descriptor` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. This value is set in seconds. For more information, see "session-descriptor" on page B-7.
- Set the `session-timeout` element in the J2EE standard Web application deployment descriptor, `web.xml`.

Configuring WebLogic Server Session Cookies

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

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

You can configure session-tracking parameters of cookies in the WebLogic-specific deployment descriptor, `weblogic.xml`. A complete list of session and cookie-related parameters is available in "session-descriptor" on page B-7.

Configuring Application Cookies That Outlive a Session

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

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

You can set the maximum life of a cookie with the `cookie-max-age-secs` element in the session descriptor of the `weblogic.xml` deployment descriptor. See "`cookie-max-age-secs`" on page B-11.

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 "Logging Out and Ending a Session" on page 8-18.

Enabling Web applications to share the same session

By default, Web applications do not share the same session. If you would like Web applications to share the same session, you can configure the session descriptor at the application level in the `weblogic-application.xml` deployment descriptor. To enable Web applications to share the same session, set the `sharing-enabled` attribute in the session descriptor to `true` in the `weblogic-application.xml` deployment descriptor. See "sharing-enabled" in "session-descriptor" on page B-7.

The session descriptor configuration that you specify at the application level overrides any session descriptor configuration that you specify at the Web application level for all of the Web applications in the application. If you set the `sharing-enabled` attribute to true at the Web application level, it will be ignored.

All Web applications in an application are automatically started using the same session instance if you specify the session descriptor in the `weblogic-application.xml` deployment descriptor and set the `sharing-enabled` attribute to `true` as in the following example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<weblogic-application xmlns="http://www.bea.com/ns/weblogic/90";;>
   ...
 <session-descriptor>
     <persistent-store-type>memory</persistent-store-type>
     <sharing-enabled>true</sharing-enabled>
     ...
 </session-descriptor>
...
</weblogic-application>
```

Configuring Session Persistence

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

There are five different implementations of session persistence:

- Memory (single-server, non-replicated)
- File system persistence

- JDBC persistence
- Cookie-based session persistence
- In-memory replication (across a cluster)

The first four are discussed here; in-memory replication is discussed in "HTTP Session State Replication," in *Using WebLogic Server Clusters*.

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

Attributes Shared by Different Types of Session Persistence

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

`cache-size`

Limits the number of cached sessions that can be active in memory at any one time. If you expect high volumes of simultaneous active sessions, you do not want these sessions to soak up the RAM of your server because this may cause performance problems swapping to and from virtual memory. When the cache is full, the least recently used sessions are stored in the persistent store and recalled automatically when required. If you do not use persistence, this property is ignored, and there is no soft limit to the number of sessions allowed in main memory. By default, the number of cached sessions is 1028. To turn off caching, set this to 0. See "cache-size" on page B-9.

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

`invalidation-interval-secs`

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 element to tune WebLogic Server for best performance on high traffic sites. See "`invalidation-interval-secs`" on page B-8.

The minimum value is every second (1). The maximum value is once a week (604,800 seconds). If not set, the attribute defaults to 60 seconds.

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

When you use memory-based storage, all session information is stored in memory and is lost when you stop and restart WebLogic Server. To use memory-based, single-server, non-replicated persistent storage, set the `persistent-store-type` parameter in the `session-descriptor` element in the `weblogic.xml` deployment descriptor file to `memory`. See "`persistent-store-type`" on page B-12.

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

Using File-based Persistent Storage

To configure file-based persistent storage for sessions:

1. In the deployment descriptor file `weblogic.xml`, set the `persistent-store-type` parameter in the `session-descriptor` element in the `weblogic.xml` deployment descriptor file to `file`. See "`persistent-store-type`" on page B-12.
2. Set the directory where WebLogic Server stores the sessions. See "`persistent-store-dir`" on page B-13.

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

Using a Database for Persistent Storage (JDBC persistence)

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

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

Configuring JDBC-based Persistent Storage

To configure JDBC-based persistent storage for sessions:

1. Set the `persistent-store-type` parameter in the `session-descriptor` element in the `weblogic.xml` deployment descriptor file to `jdbc`. See "`persistent-store-type`" on page B-12.
2. Set a JDBC connection pool to be used for persistence storage with the `persistent-store-pool` parameter in the `session-descriptor` element in the `weblogic.xml` deployment descriptor file. Use the name of a connection pool that is defined in the WebLogic Server Administration Console. See "`persistent-store-pool`" on page B-13.
3. Set up a database table named `wl_servlet_sessions` for JDBC-based persistence. The connection pool that connects to the database needs to have read/write access for this table.

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

   Set up column names and data types as follows.

Table 8-1 Creating wl_servlet_sessions

Column Name	Data 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`

Table 8-1 Creating wl_servlet_sessions

Column Name	Data Type
`wl_access_time`	Numeric column, 20 digits; for example, `NUMBER(20)`
`wl_max_inactive_interval`	Integer column; for example, Oracle `Integer`. Number of seconds between client requests before the session is invalidated. A negative time value indicates that the session should never time out.

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

Listing 8-1 Creating wl_servlet_sessions table with Oracle DBMS

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

Note: You can use the `jdbc-connection-timeout-secs` parameter to configure a maximum duration that JDBC session persistence should wait for a JDBC connection from the connection pool before failing to load the session data. For more information, see "`jdbc-connection-timeout-secs`" on page B-14.

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

Listing 8-2 Creating wl_servlet_sessions table with SqlServer 2000

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

If you are using Pointbase, Pointbase translates the SQL. For example, Pointbase would translate the SQL provided in Listing 8-1 as follows.

Listing 8-3 Creating wl_servlet_sessions table with Pointbase SQL Translation

```
SQL> describe wl_servlet_sessions;

WL_SERVLET_SESSIONS

WL_ID VARCHAR(100) NULLABLE: NO

WL_CONTEXT_PATH VARCHAR(100) NULLABLE: NO

WL_IS_NEW CHARACTER(1) NULLABLE: YES

WL_CREATE_TIME DECIMAL(20) NULLABLE: YES

WL_IS_VALID CHARACTER(1) NULLABLE: YES

WL_SESSION_VALUES BLOB(65535) NULLABLE: YES

WL_ACCESS_TIME DECIMAL(20) NULLABLE: YES

WL_MAX_INACTIVE_INTERVAL INTEGER(10) NULLABLE: YES

Primary Key: WL_CONTEXT_PATH

Primary Key: WL_ID
```

If you are using DB2, use the following SQL statement to create the `wl_servlet_sessions` table. Modify the SQL statement for use with your DBMS.

Listing 8-4 Creating wl_servlet_sessions table with DB2

```
CREATE TABLE WL_SERVLET_SESSIONS
(
   WL_ID VARCHAR(100) not null,
   WL_CONTEXT_PATH VARCHAR(100) not null,
   WL_IS_NEW SMALLINT,
   WL_CREATE_TIME DECIMAL(16),
   WL_IS_VALID SMALLINT,
   wl_session_values BLOB(10M) NOT LOGGED,
   WL_ACCESS_TIME DECIMAL(16),
   WL_MAX_INACTIVE_INTERVAL INTEGER,
   PRIMARY KEY (WL_ID,WL_CONTEXT_PATH)
);
```

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

Listing 8-5 Creating wl_servlet_sessions table with Sybase

```
create table WL_SERVLET_SESSIONS (
WL_ID                       varchar(100)                not null  ,
WL_CONTEXT_PATH             varchar(100)                not null  ,
WL_IS_NEW                   smallint                        null  ,
WL_CREATE_TIME              decimal(16,0)                   null  ,
WL_IS_VALID                 smallint                        null  ,
WL_SESSION_VALUES           image                           null  ,
WL_ACCESS_TIME              decimal(16,0)                   null  ,
WL_MAX_INACTIVE_INTERVAL    int                             null  ,
)
go
```

```
alter table WL_SERVLET_SESSIONS
add PRIMARY KEY CLUSTERED (WL_ID, WL_CONTEXT_PATH)
go
```

Caching and Database Updates for JDBC Session Persistence

WebLogic Server does not write the HTTP session state to disk if the request is read-only, meaning the request does not modify the HTTP session. Only the `wl_access_time` column is updated in the database, if the session is accessed.

For non read-only requests, the Web application container updates the database for the changes to session state after every HTTP request. This is done so that any server in the cluster can handle requests upon failovers and retrieve the latest session state from the database.

To prevent multiple database queries, WebLogic Server caches recently used sessions. Recently used sessions are not refreshed from the database for every request. The number of sessions in cache is governed by the `cache-size` parameter in the `session-descriptor` element of the WebLogic Server-specific deployment descriptor, `weblogic.xml`. See "cache-size" on page B-9.

Using Cookie-Based Session Persistence

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

There are some limitations to cookie-based session persistence:

- You can store only string attributes in the session. If you store any other type of object in the session, an `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.
- You cannot use commas (,) in a string when using cookie-based session persistence or an exception occurs.

To set up cookie-based session persistence:

1. Set the `persistent-store-type` parameter in the `session-descriptor` element in the `weblogic.xml` deployment descriptor file to `cookie`. See "`persistent-store-type`" on page B-12.
2. Optionally, set a name for the cookie using the `persistent-store-cookie-name` element. The default is `WLCOOKIE`. See "`persistent-store-cookie-name`" on page B-12.

Using URL Rewriting Instead of Cookies

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

Enable URL rewriting in WebLogic Server by setting the `url-rewriting-enabled` parameter in the WebLogic-specific deployment descriptor, `weblogic.xml`, under the `session-descriptor` element. The default value for this attribute is `true`. See "`url-rewriting-enabled`" on page B-14.

Coding Guidelines for URL Rewriting

Here are general guidelines for supporting URL rewriting.

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

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

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

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

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

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

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

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

  When a plug-in is used (Apache, NSAPI, ISAPI, `HttpClusterServlet`, or `HttpProxyServlet`) and URL rewriting is used at the back-end server using `response.sendRedirect(url)` or `response.encodeRedirectURL(url)`, then the `PathTrim` and `PathPrepend` parameters will be applied to the URL under the following condition: `PathTrim` will only be applied to the URL if `PathPrepend` is null or `PathPrepend` has been applied.

- 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.

Note: The CISCO Local Director load balancer expects a question mark "?" delimiter for URL rewriting. Because the WLS URL-rewriting mechanism uses a semicolon ";" as the delimiter, our URL re-writing is incompatible with this load balancer.

URL Rewriting and Wireless Access Protocol (WAP)

If you are writing a WAP application, you must use URL rewriting because the WAP protocol does not support cookies. In addition, some WAP devices have a 128-character limit on the length of a URL (including attributes), which limits the amount of data that can be transmitted using URL rewriting. To allow more space for attributes, you can limit the size of the session ID that is randomly generated by WebLogic Server.

In particular, to use the `WAPEnabled` attribute, use the Administration Console at Server→Protocols→HTTP→Advanced Options. The `WAPEnabled` attribute restricts the size of the session ID to 52 characters and disallows special characters, such as ! and #. You can also use the IDLength parameter of weblogic.xml to further restrict the size of the session ID. For additional details, see "`id-length`" on page B-9.

Session Tracking from a Servlet

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.

The following sections discuss various aspects of tracking sessions from an HTTP servlet:

A History of Session Tracking

Before session tracking matured conceptually, developers tried to build state into their pages by stuffing information into hidden fields on a page or embedding user choices into URLs used in links with a long string of appended characters. You can see good examples of this at most search engine sites, many of which still depend on CGI. These sites track user choices with URL

parameter *name=value* pairs that are appended to the URL, after the reserved HTTP character `?`. This practice can result in a very long URL that the CGI script must carefully parse and manage. The problem with this approach is that you cannot pass this information from session to session. Once you lose control over the URL—that is, once the user leaves one of your pages—the user information is lost forever.

Later, Netscape introduced browser *cookies*, which enable you to store user-related information about the client for each server. However, some browsers still do not fully support cookies, and some users prefer to turn off the cookie option in their browsers. Another factor that should be considered is that most browsers limit the amount of data that can be stored with a cookie.

Unlike the CGI approach, the HTTP servlet specification defines a solution that allows the server to store user details on the server beyond a single session, and protects your code from the complexities of tracking sessions. Your servlets can use an `HttpSession` object to track a user's input over the span of a single session and to share session details among multiple servlets. Session data can be persisted using a variety of methods available with WebLogic Service.

Tracking a Session with an HttpSession Object

According to the Java Servlet API, which WebLogic Server implements and supports, each servlet can access a server-side session by using its `HttpSession` object. You can access an `HttpSession` object in the `service()` method of the servlet by using the `HttpServletRequest` object with the variable `request` variable, as shown:

```
HttpSession session = request.getSession(true);
```

An `HttpSession` object is created if one does not already exist for that client when the `request.getSession(true)`method is called with the argument `true`. The session object lives on WebLogic Server for the lifetime of the session, during which the session object accumulates information related to that client. Your servlet adds or removes information from the session object as necessary. A session is associated with a particular client. Each time the client visits your servlet, the same associated `HttpSession` object is retrieved when the `getSession()` method is called.

For more details on the methods supported by the `HttpSession`, refer to the HttpServlet API at `http://java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/http/HttpSession.html`.

In the following example, the `service()` method counts the number of times a user requests the servlet during a session.

```
public void service(HttpServletRequest request,
                    HttpServletResponse, response)
```

```
            throws IOException
{
  // Get the session and the counter param attribute
  HttpSession session = request.getSession (true);
  Integer ival = (Integer)
                session.getAttribute("simplesession.counter");
  if (ival == null) // Initialize the counter
    ival = new Integer (1);
  else // Increment the counter
    ival = new Integer (ival.intValue () + 1);
  // Set the new attribute value in the session
  session.setAttribute("simplesession.counter", ival);
  // Output the HTML page
  out.print("<HTML><body>");
  out.print("<center> You have hit this page ");
  out.print(ival + " times!");
  out.print("</body></html>");
}
```

Lifetime of a Session

A session tracks the selections of a user over a series of pages in a single transaction. A single transaction may consist of several tasks, such as searching for an item, adding it to a shopping cart, and then processing a payment. A session is transient, and its lifetime ends when one of the following occurs:

- A user leaves your site and the user's browser does not accept cookies.
- A user quits the browser.
- The session is timed out due to inactivity.
- The session is completed and invalidated by the servlet.
- The user logs out and is invalidated by the servlet.

For more persistent, long-term storage of data, your servlet should write details to a database using JDBC or EJB and associate the client with this data using a long-lived cookie and/or username and password. Although this document states that sessions use cookies and persistence internally, *you should not use sessions as a general mechanism for storing data about a user.*

How Session Tracking Works

How does WebLogic Server know which session is associated with each client? When an `HttpSession` is created in a servlet, it is associated with a unique ID. The browser must provide this session ID with its request in order for the server to find the session data again. The server attempts to store this ID by setting a cookie on the client. Once the cookie is set, each time the browser sends a request to the server it includes the cookie containing the ID. The server automatically parses the cookie and supplies the session data when your servlet calls the `getSession()` method.

If the client does not accept cookies, the only alternative is to encode the ID into the URL links in the pages sent back to the client. For this reason, you should always use the `encodeURL()` method when you include URLs in your servlet response. WebLogic Server detects whether the browser accepts cookies and does not unnecessarily encode URLs. WebLogic automatically parses the session ID from an encoded URL and retrieves the correct session data when you call the `getSession()` method. Using the `encodeURL()` method ensures no disruption to your servlet code, regardless of the procedure used to track sessions. For more information, see "Using URL Rewriting Instead of Cookies" on page 8-12.

Detecting the Start of a Session

After you obtain a session using the `getSession(true)` method, you can tell whether the session has just been created by calling the `HttpSession.isNew()` method. If this method returns `true`, then the client does not already have a valid session, and at this point it is unaware of the new session. The client does not become aware of the new session until a reply is posted back from the server.

Design your application to accommodate new or existing sessions in a way that suits your business logic. For example, your application might redirect the client's URL to a login/password page if you determine that the session has not yet started, as shown in the following code example:

```
HttpSession session = request.getSession(true);
if (session.isNew()) {
  response.sendRedirect(welcomeURL);
}
```

On the login page, provide an option to log in to the system or create a new account. You can also specify a login page in your Web Application using the `login-config` element of the J2EE standard Web application deployment descriptor, `web.xml`.

Setting and Getting Session Name/Value Attributes

You can store data in an `HttpSession` object using *`name=value`* pairs. Data stored in a session is available through the session. To store data in a session, use these methods from the `HttpSession` interface:

```
getAttribute()
getAttributeNames()
setAttribute()
removeAttribute()
```

The following code fragment shows how to get all the existing *`name=value`* pairs:

```
Enumeration sessionNames = session.getAttributeNames();
String sessionName = null;
Object sessionValue = null;

while (sessionNames.hasMoreElements()) {
  sessionName = (String)sessionNames.nextElement();
  sessionValue = session.getAttribute(sessionName);
  System.out.println("Session name is " + sessionName +
                     ", value is " + sessionValue);
}
```

To add or overwrite a named attribute, use the `setAttribute()` method. To remove a named attribute altogether, use the `removeAttribute()` method.

Note: You can add any Java descendant of `Object` as a session attribute and associate it with a name. However, if you are using session persistence, your attribute *`value`* objects must implement `java.io.Serializable`.

Logging Out and Ending a Session

If your application deals with sensitive information, consider offering the ability to log out of the session. This is a common feature when using shopping carts and Internet email accounts. When the same browser returns to the service, the user must log back in to the system.

Using session.invalidate() for a Single Web Application

User authentication information is stored both in the users's session data and in the context of a server or virtual host that is targeted by a Web Application. Using 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.

Do not reference an invalidated session after calling `session.invalidate()`. If you do, an `IllegalStateException` is thrown. The next time a user visits your servlet from the same browser, the session data will be missing, and a new session will be created when you call the `getSession(true)` method. At that time you can send the user to the login page again.

Implementing Single Sign-On for Multiple Applications

If the server or virtual host is targeted by many Web Applications, another means is required to log out a user from all Web Applications. Because the Servlet specification does not provide an API for logging out a user from all Web Applications, the following methods are provided.

`weblogic.servlet.security.ServletAuthentication.logout()`
: Removes the authentication data from the users's session data, which logs out a user but allows the session to remain alive.

`weblogic.servlet.security.ServletAuthentication.invalidateAll()`
: Invalidates all the sessions and removes the authentication data for the current user. The cookie is also invalidated.

`weblogic.servlet.security.ServletAuthentication.killCookie()`
: Invalidates the current cookie by setting the cookie so that it expires immediately when the response is sent to the browser. This method depends on a successful response reaching the user's browser. The session remains alive until it times out.

Exempting a Web Application for Single Sign-on

If you want to exempt a Web Application from participating in single sign-on, define a different cookie name for the exempted Web Application. For more information, see "Configuring WebLogic Server Session Cookies" on page 8-2.

Configuring Session Tracking

WebLogic Server provides many configurable attributes that determine how WebLogic Server handles session tracking. For details about configuring these session tracking attributes, see "session-descriptor" on page B-7.

Using URL Rewriting Instead of Cookies

In some situations, a browser may not accept cookies, which means that session tracking with cookies is not possible. URL rewriting is a workaround to this scenario 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 hyperlinks 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 and finds the appropriate `HttpSession`. Then you use the `getSession()` method to access session data.

To enable URL rewriting in WebLogic Server, set the `URL-rewriting-enabled` parameter to `true` in the `session-descriptor` element of the WebLogic Server-specific deployment descriptor, `weblogic.xml`. See "`url-rewriting-enabled`" on page B-14.

To make sure your code correctly handles URLs in order to support URL rewriting, consider the following guidelines:

- You should 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 necessary, rewrites the URL by including the session ID in the URL.

- Encode URLs that send redirects, as well as URLs that are returned as a response to WebLogic Server. For example:

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

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

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

Note: The CISCO Local Director load balancer expects a question mark "?" delimiter for URL rewriting. Because the WLS URL-rewriting mechanism uses a semicolon ";" as the delimiter, our URL re-writing is incompatible with this load balancer.

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 impose a 128-character limit (including parameters) on the length of a URL, 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 by specifying the number of bytes with the `id-length` parameter in the `session-descriptor` element of the WebLogic Server-specific deployment descriptor, `weblogic.xml`. See "`id-length`" on page B-9.

The minimum value is 8 bytes; the default value is 52 bytes; the maximum value is `Integer.MAX_VALUE`.

Making Sessions Persistent

You can set up WebLogic Server to record session data in a persistent store. If you are using session persistence, you can expect the following characteristics:

- Good failover, because sessions are saved when servers fail.
- Better load balancing, because any server can handle requests for any number of sessions, and use caching to optimize performance. For more information, see the `cache-size` property, at "Configuring Session Persistence" on page 8-4.
- Sessions can be shared across clustered WebLogic Servers. Note that session persistence is no longer a requirement in a WebLogic Cluster. Instead, you can use in-memory replication of state. For more information, see *Using WebLogic Server Clusters*.
- For customers who want the highest in servlet session persistence, JDBC-based persistence is the best choice. For customers who want to sacrifice some amount of session persistence in favor of drastically better performance, in-memory replication is the appropriate choice. JDBC-based persistence is noticeably slower than in-memory replication. In some cases,

in-memory replication has outperformed JDBC-based persistence for servlet sessions by a factor of eight.

- You can put any kind of Java object into a session, but for file, JDBC, and in-memory replication, only objects that are `java.io.Serializable` can be stored in a session. For more information, see "Configuring Session Persistence" on page 8-4.

Scenarios to Avoid When Using Sessions

Do not use session persistence for storing long-term data between sessions. In other words, do not rely on a session still being active when a client returns to a site at some later date. Instead, your application should record long-term or important information in a database.

Sessions are not a convenience wrapper around cookies. Do not attempt to store long-term or limited-term client data in a session. Instead, your application should create and set its own cookies on the browser. Examples include an auto-login feature that allows a cookie to live for a long period, or an auto-logout feature that allows a cookie to expire after a short period of time. Here, you should not attempt to use HTTP sessions. Instead, you should write your own application-specific logic.

Use Serializable Attribute Values

When you use persistent sessions, all attribute `value` objects that you add to the session must implement `java.io.Serializable`. For more details on writing serializable classes, refer to the online java tutorial about serializable objects at `http://java.sun.com/docs/books/tutorial/essential/io/providing.html`.

If you add your own serializable classes to a persistent session, make sure that each instance variable of your class is also serializable. Otherwise, you can declare it as `transient`, and WebLogic Server does not attempt to save that variable to persistent storage. One common example of an instance variable that must be made `transient` is the `HttpSession` object. (See the notes on using serialized objects in sessions in the section "Making Sessions Persistent" on page 8-21.)

The `HttpServletRequest`, `ServletContext`, and `HttpSession` attributes will be serialized when a WebLogic Server instance detects a change in the Web application classloader. The classloader changes when a Web application is redeployed, when there is a dynamic change in a servlet, or when there is a cross Web application forward or include.

To avoid having the attribute serialized, during a dynamic change in a servlet, turn off `servlet-reload-check-secs` in `weblogic.xml`. There is no way to avoid serialization of

attributes for cross Web application dispatch or redeployment. See "servlet-reload-check-secs" on page B-18.

Configuring Session Persistence

For details about setting up persistent sessions, see "Configuring Session Persistence" on page 8-4.

Configuring a Maximum Limit on In-memory Servlet Sessions

Without the ability to configure in-memory servlet session use, as new sessions are continually created, the server eventually throws out of memory. To protect against this, WebLogic Server provides a configurable bound on the number of sessions created. When this number is exceeded, the `weblogic.servlet.SessionCreationException` occurs for each attempt to create a new session. This feature applies to both replicated and non-replicated in-memory sessions.

To configure bound in-memory servlet session use, you set the limitation in the `max-in-memory-sessions` element in the `weblogic.xml` deployment descriptor. See "max-in-memory-sessions" on page B-10.

Enabling Session Memory Overload Protection

When memory is overloaded, a `weblogic.servlet.SessionCreationException` (`RuntimeException`) for any `getSession(true)` attempts occurs. As the person developing the servlet, you should handle this exception as follows:

- Return the appropriate error message to the user when the exception occurs, explaining the situation.
- Map `weblogic.servlet.SessionCreationException` to an error page in the J2EE standard Web Application deployment descriptor, `web.xml`.

By default, memory overload protection is turned off. You can enable it with a domain-level flag:

```
weblogic.management.configuration.WebAppContainerMBean.OverloadProtectionEnabled
```

CHAPTER 9

Application Events and Event Listener Classes

The following sections discuss application events and event listener classes:

Overview of Application Event Listener Classes

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

The following is an overview of this process:

1. The user creates an event listener class that implements one of the listener interfaces.
2. This implementation is registered in the deployment descriptor.

3. At deployment time, the servlet container constructs an instance of the event listener class. (This is why the public constructor must exist, as discussed in "Writing an Event Listener Class" on page 9-5.)
4. At runtime, the servlet container invokes on the instance of the listener class.

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

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

Use Web application event listener classes to:

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

Servlet Context Events

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

Table 9-1 Servlet Context Events

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()`

Table 9-1 Servlet Context Events

Type of Event	Interface	Method
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 your event listener class must implement to respond to the HTTP session events and the methods invoked when the HTTP session events occur.

Table 9-2 HTTP Session Events

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.HttpSessionAttributeListener`	`attributeAdded()`
An attribute is removed.	`javax.servlet.http.HttpSessionAttributeListener`	`attributeRemoved()`
An attribute is replaced.	`javax.servlet.http.HttpSessionAttributeListener`	`attributeReplaced()`

Note: The Servlet 2.4 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.

Servlet Request Events

The following table lists the types of Servlet request events, the interface your event listener class must implement to manage state across the lifecycle of servlet requests and the methods invoked when the request events occur.

Table 9-3 Servlet Request Events

Type of Event	Interface	Method
The request is about to go out of scope of the Web application.	`javax.servlet.ServletRequestListener`	`requestDestroyed()`
The request is about to come into scope of the Web application.	`javax.servlet.ServletRequestListener`	`requestInitialized()`
Notification that a new attribute was added to the servlet request. Called after the attribute is added.	`javax.servlet.ServletRequestAttributeListener`	`attributeAdded()`
Notification that a new attribute was removed from the servlet request. Called after the attribute is removed.	`javax.servlet.ServletRequestAttributeListener`	`attributeRemoved()`
Notification that an attribute was replaced on the servlet request. Called after the attribute is replaced.	`javax.servlet.ServletRequestAttributeListener`	`attributeReplaced()`

Configuring an Event Listener Class

To configure an event listener class:

1. Open the `web.xml` deployment descriptor of the Web application for which you are creating an event listener class in a text editor. The `web.xml` file is located in the `WEB-INF` directory of your Web application.

2. Add an event declaration using the `listener` element of the `web.xml` deployment descriptor. The event declaration defines the event listener class that is invoked when the event occurs. The `listener` element must directly follow the `filter` and `filter-mapping` elements and directly precede the `servlet` element. You can specify more than one event listener class for each type of event. WebLogic Server invokes the event listener classes in the order that they appear in the deployment descriptor (except for shutdown events, which are invoked in the reverse order). For example:

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

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

3. Write and deploy the event listener class. For details, see the section, "Writing an Event Listener Class" on page 9-5.

Writing an Event Listener Class

To write an event listener class:

1. Create a new event listener class that implements the appropriate interface for the type of event to which your class responds. For a list of these interfaces, see "Servlet Context Events" on page 9-2 or "HTTP Session Events" on page 9-3. See "Templates for Event Listener Classes" on page 9-6 for sample templates you can use to get started.

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

```
public class MyListener {

// public constructor

public MyListener() { /* ... */ }

}
```

3. Implement the required methods of the interface. See the J2EE API Reference (Javadocs) at `http://java.sun.com/j2ee/tutorial/api/index.html` for more information.

4. Copy the compiled event listener classes into the `WEB-INF/classes` directory of the Web application, or package them into a JAR file and copy the JAR file into the `WEB-INF/lib` directory of the Web application.

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

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

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

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

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

Templates for Event Listener Classes

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

Servlet Context Event Listener Class Example

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

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

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

    }
    public void contextDestroyed(ServletContextEvent event) {

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

    }
}
```

HTTP Session Attribute Event Listener Class Example

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

public final class MySessionAttributeListenerClass implements
   HttpSessionAttributeListener {

    public void attributeAdded(HttpSessionBindingEvent sbe) {
      /* This method is called when an attribute
         is added to a session.
      */
    }
    public void attributeRemoved(HttpSessionBindingEvent sbe) {
      /* This method is called when an attribute
         is removed from a session.
      */
    }
    public void attributeReplaced(HttpSessionBindingEvent sbe) {
      /* This method is invoked when an attibute
         is replaced in a session.
      */
    }
}
```

Additional Resources

- Servlet 2.4 Specification from Sun Microsystems at `http://java.sun.com/products/servlet/download.html#specs`
- 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 10

WebLogic JSP Reference

The following sections provide reference information for writing JavaServer Pages (JSPs):

- "Using the WebLogic JSP Compiler" on page 10-23

JSP Tags

The following table describes the basic tags that you can use in a JSP page. Each shorthand tag has an XML equivalent.

Table 10-1 Basic Tags for JSP Pages

JSP Tag	Syntax	Description

`Scriptlet`	`<% java_code %>`
	. . . or use the XML equivalent:
	`<jsp:scriptlet>`
	`  java_code`
	`</jsp:scriptlet>`
		Embeds Java source code scriptlet in your HTML page. The Java code is executed and its output is inserted in sequence with the rest of the HTML in the page. For details, see "Scriptlets" on page 10-6.

`Directive`	`<%@ dir-type dir-attr %>`
	. . . or use the XML equivalent:
	`<jsp:directive.dir_type dir_attr />`
		Directives contain messages to the application server.
		A directive can also contain name/value pair attributes in the form `attr="value"`, which provides additional instructions to the application server. See "Directives for WebLogic JSP" on page 10-5.

`Declarations`	`<%! declaration %>`
	. . . or use XML equivalent...
	`<jsp:declaration>`
	`  declaration;`
	`</jsp:declaration>`
		Declares a variable or method that can be referenced by other declarations, scriptlets, or expressions in the page. See "Declarations" on page 10-6.

`Expression`	`<%= expression %>`
	. . . or use XML equivalent...
	`<jsp:expression>`
	`expression`
	`</expression>`
		Defines a Java expression that is evaluated at page request time, converted to a `String`, and sent inline to the output stream of the JSP response. See "Expressions" on page 10-7.

Table 10-1 Basic Tags for JSP Pages

JSP Tag	Syntax	Description

`Actions`	`<jsp:useBean ... >`
	JSP body is included if the bean is instantiated here
	`</jsp:useBean>`
	`<jsp:setProperty ... >`
	`<jsp:getProperty ... >`
	`<jsp:include ... >`
	`<jsp:forward ... >`
	`<jsp:plugin ... >`
		Provide access to advanced features of JSP, and only use XML syntax. These actions are supported as defined in the JSP 2.0 specification. See "Actions" on page 10-9.

`Comments`	<%/* `comment` */%>	Ensure that your comments are removed from the viewable source of your HTML files by using only JSP comment tags. HTML comments remain visible when the user selects view source in the browser.

Reserved Words for Implicit Objects

JSP reserves words for implicit objects in scriptlets and expressions. These implicit objects represent Java objects that provide useful methods and information for your JSP page. WebLogic JSP implements all implicit objects defined in the JSP 2.0 specification. The JSP API is described in the Javadocs available from the Sun Microsystems JSP Home Page at `http://www.java.sun.com/products/jsp/index.html`.

Note: Use these implicit objects only within scriptlets or expressions. Using these keywords from a method defined in a declaration causes a translation-time compilation error because such usage causes your page to reference an undefined variable.

`request`

`request` represents the `HttpServletRequest` object. It contains information about the request from the browser and has several useful methods for getting cookie, header, and session data.

`response`

`response` represents the `HttpServletResponse` object and several useful methods for setting the response sent back to the browser from your JSP page. Examples of these responses include cookies and other header information.

Warning: You cannot use the `response.getWriter()` method from within a JSP page; if you do, a run-time exception is thrown. Use the `out` keyword to send the JSP response back to the browser from within your scriptlet code whenever possible. The WebLogic Server implementation of `javax.servlet.jsp.JspWriter` uses `javax.servlet.ServletOutputStream`, which implies that you *can* use `response.getServletOutputStream()`. Keep in mind, however, that this implementation is specific to WebLogic Server. To keep your code maintainable and portable, use the `out` keyword.

`out`

`out` is an instance of `javax.jsp.JspWriter` that has several methods you can use to send output back to the browser.

If you are using a method that requires an output stream, then `JspWriter` does not work. You can work around this limitation by supplying a buffered stream and then writing this stream to `out`. For example, the following code shows how to write an exception stack trace to `out`:

```
ByteArrayOutputStream ostr = new ByteArrayOutputStream();
exception.printStackTrace(new PrintWriter(ostr));
out.print(ostr);
```

`pageContext`

`pageContext` represents a `javax.servlet.jsp.PageContext` object. It is a convenience API for accessing various scoped namespaces and servlet-related objects, and provides wrapper methods for common servlet-related functionality.

`session`

`session` represents a `javax.servlet.http.HttpSession` object for the request. The session directive is set to true by default, so the `session` is valid by default. The JSP 2.0 specification states that if the session directive is set to `false`, then using the `session` keyword results in a fatal translation time error.

`application`

`application` represents a `javax.servlet.ServletContext` object. Use it to find information about the servlet engine and the servlet environment.

When forwarding or including requests, you can access the servlet `requestDispatcher` using the `ServletContext`, or you can use the JSP `forward` directive for forwarding requests to other servlets, and the JSP `include` directive for including output from other servlets.

`config`

`config` represents a `javax.servlet.ServletConfig` object and provides access to the servlet instance initialization parameters.

`page`

`page` represents the servlet instance generated from this JSP page. It is synonymous with the Java keyword `this` when used in your scriptlet code.

To use `page`, you must cast it to the class type of the servlet that implements the JSP page, because it is defined as an instance of `java.lang.Object`. By default, the servlet class is named after the JSP filename. For convenience, we recommend that you use the Java keyword `this` to reference the servlet instance and get access to initialization parameters, instead of using `page`.

Directives for WebLogic JSP

Use directives to instruct WebLogic JSP to perform certain functions or interpret the JSP page in a particular way. You can insert a directive anywhere in a JSP page. The position is generally irrelevant (except for the `include` directive), and you can use multiple directive tags. A directive consists of a directive type and one or more attributes of that type.

You can use either of two types of syntax: shorthand or XML:

- Shorthand:

  ```
  <%@ dir_type dir_attr %>
  ```

- XML:

  ```
  <jsp:directive.dir_type dir_attr />
  ```

Replace `dir_type` with the directive type, and `dir_attr` with a list of one or more directive attributes for that directive type.

There are three types of directives `page`, `taglib`, or `include`.

Using the page Directive to Set Character Encoding

To specify a character encoding set, use the following directive at the top of the page:

```
<%@ page contentType="text/html; charset=custom-encoding” %>
```

The character set you specify with a `contentType` directive specifies the character set used in the JSP as well as any JSP *included* in that JSP.

You can specify a default character encoding by specifying it in the WebLogic-specific deployment descriptor for your Web Application.

Using the taglib Directive

Use a `taglib` directive to declare that your JSP page uses custom JSP tag extensions that are defined in a tag library. For details about writing and using custom JSP tags, see "Programming WebLogic JSP Extensions."

Declarations

Use declarations to define variables and methods at the class-scope level of the generated JSP servlet. Declarations made between JSP tags are accessible from other declarations and scriptlets in your JSP page. For example:

```
<%!
  int i=0;
  String foo= "Hello";
  private void bar() {
    // ...java code here...
  }
%>
```

Remember that class-scope objects are shared between multiple threads being executed in the same instance of a servlet. To guard against sharing violations, synchronize class scope objects. If you are not confident writing thread-safe code, you can declare your servlet as not-thread-safe by including the following directive:

```
<%@ page isThreadSafe="false" %>
```

By default, this attribute is set to `true`. Setting `isThreadSafe` to `false` consumes additional memory and can cause performance to degrade.

Scriptlets

JSP scriptlets make up the Java body of your JSP servlet's HTTP response. To include a scriptlet in your JSP page, use the shorthand or XML scriptlet tags shown here:

Shorthand:

```
<%
  // Your Java code goes here
%>
```

XML:

```
<jsp:scriptlet>
  // Your Java code goes here
</jsp:scriptlet>
```

Note the following features of scriptlets:

- You can have multiple blocks of scriptlet Java code mixed with plain HTML.
- You can switch between HTML and Java code anywhere, even within Java constructs and blocks. In "Example of a JSP with HTML and Embedded Java" on page 8 the example declares a Java loop, switches to HTML, and then switches back to Java to close the loop. The HTML within the loop is generated as output multiple times as the loop iterates.
- You can use the predefined variable `out` to print HTML text directly to the servlet output stream from your Java code. Call the `print()` method to add a string to the HTTP page response.

  Any time you print data that a user has previously supplied, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. For more information, refer to "JSP Expression Language" on page 10-12.
- The Java tag is an *inline* tag; it does not force a new paragraph.

Expressions

To include an expression in your JSP file, use the following tag:

<%= expr %>

Replace *expr* with a Java expression. When the expression is evaluated, its `string` representation is placed inline in the HTML response page. It is shorthand for

```
<% out.print( expr ); %>
```

This technique enables you to make your HTML more readable in the JSP page. Note the use of the expression tag in the example in the next section.

Expressions are often used to return data that a user has previously supplied. Any time you print user-supplied data, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. For more information, refer to "JSP Expression Language" on page 10-12.

Example of a JSP with HTML and Embedded Java

The following example shows a JSP with HTML and embedded Java:

```
<html>
  <head><title>Hello World Test</title></head>
<body bgcolor=#ffffff>
<center>
<h1> <font color=#DB1260> Hello World Test </font></h1>
<font color=navy>
<%
    out.print("Java-generated Hello World");
%>
</font>
<p> This is not Java!
<p><i>Middle stuff on page</i>
<p>
<font color=navy>
<%
     for (int i = 1; i<=3; i++) {
%>
        <h2>This is HTML in a Java loop! <%= i %> </h2>
<%
     }
%>
</font>
</center>
</body>
</html>
```

After the code shown here is compiled, the resulting page is displayed in a browser as follows:

Actions

You use JSP actions to modify, use, or create objects that are represented by JavaBeans. Actions use XML syntax exclusively.

Using JavaBeans in JSP

The `<jsp:useBean>` action tag allows you to instantiate Java objects that comply with the JavaBean specification, and to refer to them from your JSP pages.

To comply with the JavaBean specification, objects need:

- A public constructor that takes no arguments
- A `setVariable()` method for each `variable` field
- A `getVariable()` method for each `variable` field

Instantiating the JavaBean Object

The <jsp:useBean> tag attempts to retrieve an existing named Java object from a specific scope and, if the existing object is not found, may attempt to instantiate a new object and associate it with the name given by the id attribute. The object is stored in a location given by the scope attribute, which determines the availability of the object. For example, the following tag attempts to retrieve a Java object of type examples.jsp.ShoppingCart from the HTTP session under the name cart.

```
<jsp:useBean id="cart"
    class="examples.jsp.ShoppingCart" scope="session"/>
```

If such an object does not currently exist, the JSP attempts to create a new object, and stores it in the HTTP session under the name cart. The class should be available in the CLASSPATH used to start WebLogic Server, or in the WEB-INF/classes directory of the Web Application containing the JSP.

It is good practice to use an errorPage directive with the <jsp:useBean> tag because there are run-time exceptions that must be caught. If you do not use an errorPage directive, the class referenced in the JavaBean cannot be created, an InstantiationException is thrown, and an error message is returned to the browser.

You can use the type attribute to cast the JavaBean type to another object or interface, provided that it is a legal type cast operation within Java. If you use the attribute without the class attribute, your JavaBean object must already exist in the scope specified. If it is not legal, an InstantiationException is thrown.

Doing Setup Work at JavaBean Instantiation

The <jsp:useBean> tag syntax has another format that allows you to define a body of JSP code that is executed when the object is instantiated. The body is not executed if the named JavaBean already exists in the specified scope. This format allows you to set up certain properties when the object is first created. For example:

```
<jsp:useBean id="cart" class="examples.jsp.ShoppingCart"
 scope=session>
    Creating the shopping cart now...
    <jsp:setProperty name="cart"
     property="cartName" value="music">
</jsp:useBean>
```

Note: If you use the `type` attribute without the `class` attribute, a JavaBean object is never instantiated, and you should not attempt to use the tag format to include a body. Instead, use the single tag format. In this case, the JavaBean must exist in the specified scope, or an `InstantiationException` is thrown. Use an `errorPage` directive to catch the potential exception.

Using the JavaBean Object

After you instantiate the JavaBean object, you can refer to it by its `id` name in the JSP file as a Java object. You can use it within scriptlet tags and expression evaluator tags, and you can invoke its `setXxx()` or `getXxx()` methods using the `<jsp:setProperty>` and `<jsp:getProperty>` tags, respectively.

Defining the Scope of a JavaBean Object

Use the `scope` attribute to specify the availability and life-span of the JavaBean object. The scope can be one of the following:

`page`
: This is the default scope for a JavaBean, which stores the object in the `javax.servlet.jsp.PageContext` of the current page. It is available only from the current invocation of this JSP page. It is not available to included JSP pages, and it is discarded upon completion of this page request.

`request`
: When the `request` scope is used, the object is stored in the current `ServletRequest`, and it is available to other included JSP pages that are passed the same request object. The object is discarded when the current request is completed.

`session`
: Use the `session` scope to store the JavaBean object in the HTTP session so that it can be tracked across several HTTP pages. The reference to the JavaBean is stored in the page's `HttpSession` object. Your JSP pages must be able to participate in a session to use this scope. That is, you must not have the `page` directive `session` set to `false`.

`application`
: At the `application`-scope level, your JavaBean object is stored in the Web Application. Use of this scope implies that the object is available to any other servlet or JSP page running in the same Web Application in which the object is stored.

For more information about using JavaBeans, see the JSP 2.0 specification at `http://www.java.sun.com/products/jsp/index.html`.

Forwarding Requests

If you are using any type of authentication, a forwarded request made with the `<jsp:forward>` tag, by default, does not require the user to be re-authenticated. You can change this behavior to require authentication of a forwarded request by adding the `<check-auth-on-forward/>` element to the `<container-descriptor>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. For example:

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

Including Requests

You can use the `<jsp:include>` tag to include another resource in a JSP. This tag takes two attributes:

page

Use the page attribute to specify the included resource. For example:

```
<jsp:include page="somePage.jsp"/>
```

flush

Setting this boolean attribute to `true` buffers the page output and then flushes the buffer before including the resource.

Setting `flush="false"` can be useful when the `<jsp:include>` tag is located within another tag on the JSP page and you want the included resource to be processed by the tag.

JSP Expression Language

The new JSP expression language (JSP EL) is inspired by both ECMAScript and the XPath expression languages. The JSP EL is available in attribute values for standard and custom actions and within template text. In both cases, the JSP EL is invoked consistently by way of the construct `${expr}`.

The addition of the JSP EL to the JSP technology better facilitates the writing of scriptlets JSP pages. These pages can use JSP EL expressions but cannot use Java scriptlets, Java expressions, or Java declaration elements. You can enforce this usage pattern through the `scripting-invalid` JSP configuration element of the `web.xml` deployment descriptor.

For more information on the JSP expression language, see the JSP 2.0 specification.

Expressions and Attribute Values

You can use JSP EL expressions in any attribute that can accept a run-time expression, whether it is a standard action or a custom action. The following are three use cases for expressions in attribute values:

- The attribute value contains a single expression construct `<some:tag value="${expr}"/>`. In this case, the expression is evaluated and the result is coerced to the attribute's expected type according to the type conversion rules described in section 2.8, "Type Conversion," of the JSP 2.0 specification.
- The attribute value contains one or more expressions separated or surrounded by text: `<some:tag value="some${expr}${expr}text${expr}"/>`. In this case, the expressions are evaluated from left to right, coerced to Strings (according to the type conversion rules described later), and concatenated with any intervening text. The resulting String is then coerced to the attributes expected type according to the type conversion rules described in section 2.8, "Type Conversion," of the JSP 2.0 specification.
- The attribute value contains only text: `<some:tag value="sometext"/>`. In this case, the attribute's String value is coerced to the attribute's expected type according to the type conversion rules described in section 2.8, "Type Conversion," of the JSP 2.0 specification.

Note: These rules are equivalent to the JSP 2.0 conversions, except that empty strings are treated differently.

The following two conditions must be satisfied when using JSPX:

- `web.xml` – The `web-app` must define the Servlet version attribute as `2.4` or higher; otherwise, all JSP EL functions are ignored.
- TLD file – Namespace declaration is required for the `jsp` prefix, as follows: `<html xmlns:jsp="http://java.sun.com/JSP/Page";`

The following shows a conditional action that uses the JSP EL to test whether a property of a bean is less than 3.

```
<c:if test="${bean1.a < 3}">

...

</c:if>
```

Note that the normal JSP coercion mechanism already allows for: `<mytags:if test="true" />`. There may be literal values that include the character sequence ${. If this is the case, a literal with that value can be used as shown here:

```
<mytags:example code="an expression is ${'${'}expr}" />
```

The resulting attribute value would then be the string an expression is `${expr}`.

Expressions and Template Text

You can use the JSP EL directly in template text; this can be inside the body of custom or standard actions or in template text outside of any action. An exception to this use is if the body of the tag is tag dependent or if the JSP EL is turned off (usually for compatibility issues) explicitly through a directive or implicitly.

The semantics of a JSP EL expression are the same as with Java expressions: the value is computed and inserted into the current output. In cases where escaping is desired (for example, to help prevent cross-site scripting attacks), you can use the JSTL core tag `<c:out>`. For example:

```
<c:out value=”${anELexpression}” />
```

The following shows a custom action where two JSP EL expressions are used to access bean properties:

```
<c:wombat>
```

One value is `${bean1.a}` and another is `${bean2.a.c}`.

```
</c:wombat>
```

JSP Expression Language Implicit Objects

There are several implicit objects that are available to JSP EL expressions used in JSP pages. These objects are always available under these names:

`pageContext`

: `pageContext` represents the `pageContext` object.

`pageScope`

: `pageContext` represents a Map that maps page-scoped attribute names to their values.

`requestScope`

: `requestScope` represents a Map that maps request-scoped attribute names to their values.

`sessionScope`

: `sessopmScope` represents a Map that maps session-scoped attribute names to their values.

`applicationScope`

`applicationScope` represents a Map that maps application-scoped attribute names to their values.

`param`

`param` represents a Map that maps parameter names to a single String parameter value (obtained by calling `ServletRequest.getParameter(String name)`).

`paramValues`

`paramValues` represents a Map that maps parameter names to a single String[] of all values for that parameter (obtained by calling `ServletRequest.getParameterValues(String name)`).

`header`

`header` represents a Map that maps header names to a single String header value `(obtained by calling ServletRequest.getHeader(string name))`.

`headerValues`

`headerValues` represents a Map that maps header names to a String[]} of all values for that header (obtained by calling `ServletRequest.getHeaders(String name)`).

`cookie`

`cookie` represents a Map that maps cookie names to a single `Cookie` object. Cookies are retrieved according to the semantics of `HttpServletRequest.getCookies()`. If the same name is shared by multiple cookies, an implementation must use the first one encountered in the array of `Cookie` objects returned by the `getCookies()` method. However, users of the cookie implicit objects must be aware that the ordering of cookies is currently unspecified in the servlet specification.

`initParam`

`initParam` represents a Map that maps context initialization parameter names to their String parameter value (obtained by calling `ServletRequest.getInitParameter(String name)`).

Table 10-1 shows some examples of using these implicit objects:

Figure 10-1 Example Uses of Implicit Objects

Expression	Result
`${pageContext.request.requestURI}`	The request's URI (obtained from `HttpServletRequest`)
`${sessionScope.profile}`	The session-scoped attribute named profile (`null` if not found)
`${param.productId}`	The String value of the `productId` parameter (`null` if not found).
`${paramValues.productId}`	The String[] containing all values of the `productId` parameter (`null` if not found).

JSP Expression Language Literals and Operators

These sections discuss JSPEL expression literals and operators. The JSP EL syntax is pretty straightforward. Variables are accessed by name. A generalized [] operator can be used to access maps, lists, arrays of objects and properties of JavaBean objects; the operator can be nested arbitrarily. The . operator can be used as a convenient shorthand for property access when the property name follows the conventions of Java identifies. However the [] operator allows for more generalized access.

Relational comparisons are allowed using the standard Java relational operators. Comparisons may be made against other values, or against boolean (for equality comparisons only), String, integer, or floating point literals. Arithmetic operators can be used to compute integer and floating point values. Logical operators are available.

Literals

Literals exist for boolean, integer, floating point, string, null.

- Boolean - true and false
- Integer - As defined by the `IntegerLiteral` construct in section 2.9, “Collected Syntax,” of the JSP 2.0 specification.
- Floating point - As defined by the `FloatingPointLiteral` construct in section 2.9, “Collected Syntax,” of the JSP 2.0 specification.

- String -With single and double quotes - " is escaped as \", ' is escaped as \', and \ is escaped as \\. Quotes only need to be escaped in a string value enclosed in the same type of quote.
- Null - null

Errors, Warnings, Default Values

JSP pages are mostly used in presentation, and in that usage, experience suggests that it is most important to be able to provide as good a presentation as possible, even when there are simple errors in the page. To meet this requirement, the JSP EL does not provide warnings, just default values and errors. Default values are typecorrect values that are assigned to a subexpression when there is some problem. An error is an exception thrown (to be handled by the standard JSP machinery).

Operators

The following is a list of operators provided by the JSP expression language:

- `.` and `[]`
- Arithmetic: +, - (binary), *, / and div, % and mod, - (unary)
- Logical: and, &&, or, ||, not, !
- Relational: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values, or against boolean, string, integer, or floating point literals.
- Empty: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
- Conditional: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.

For more information about the operators and their functions, see the JSP 2.0 specification.

Operator Precedence

The following is operator precedence, from highest to lowest, left-to-right.

- `[] .`
- `()`
- `- (unary) not ! empty`
- `* / div % mod`

- + - (binary)
- < > <= >= lt gt le ge
- == != eq ne
- && and
- || or
- ? :

JSP Expression Language Reserved Words

The following words are reserved for the language and should not be used as identifiers.

- and
- eq
- gt
- true
- instanceof
- or
- ne
- le
- false
- empty
- not
- lt
- ge
- null
- div
- mod

Note that many of these words are not in the language now, but they may be in the future, so developers should avoid using these words now.

JSP Expression Language Named Variables

A core concept in the JSP EL is the evaluation of a variable name into an object. The JSP EL API provides a generalized mechanism, a VariableResolver, that will resolve names into objects. The default resolver is what is used in the evaluation of JSP EL expressions in template and attributes. This default resolver provides the implicit objects discussed in "JSP Expression Language Implicit Objects" on page 10-14.

The default resolver also provides a map for other identifiers by looking up its value as an attribute, according to the behavior of `PageContext.findAttribute(String)` on the `pageContext` object. For example: `${product}`.

This expression looks for the attribute named product, searching the page, request, session, and application scopes, and returns its value. If the attribute is not found, null is returned. See chapter 14, "Expression Language API," of the JSP 2.0 specification. for further details on the VariableResolver and how it fits with the evaluation API.

Securing User-Supplied Data in JSPs

Expressions and scriptlets enable a JSP to receive data from a user and return the user supplied data. For example, the sample JSP in Listing 10-1 prompts a user to enter a string, assigns the string to a parameter named `userInput`, and then uses the `<%= javax.servlet.ServletRequest.getParameter("userInput")%>` expression to return the data to the browser.

Listing 10-1 Using Expressions to Return User-Supplied Content

```
<html>
  <body>
    <h1>My Sample JSP</h1>
      <form method="GET" action="mysample.jsp">
          Enter string here:
          <input type="text" name="userInput" size=50>
          <input type=submit value="Submit">
       </form>
    <br>
    <hr>
    <br>
```

```
    Output from last command:
    <%= javax.servlet.ServletRequest.getParameter("userInput")%>
  </body>
</html>
```

This ability to return user-supplied data can present a security vulnerability called **cross-site scripting**, which can be exploited to steal a user's security authorization. For a detailed description of cross-site scripting, refer to "Understanding Malicious Content Mitigation for Web Developers" (a CERT security advisory) at http://www.cert.org/tech_tips/malicious_code_mitigation.html.

To remove the security vulnerability, before you return data that a user has supplied, scan the data for any of the HTML special characters in Table 10-2. If you find any special characters, replace them with their HTML entity or character reference. Replacing the characters prevents the browser from executing the user-supplied data as HTML.

Table 10-2 HTML Special Characters that Must Be Replaced

Replace this special character:	With this entity/character reference:
<	<
>	>
(	&40;
)	&41;
#	&35;
&	&38;

Using a WebLogic Server Utility Method

WebLogic Server provides the `weblogic.servlet.security.Utils.encodeXSS()` method to replace the special characters in user-supplied data. To use this method, provide the user-supplied data as input. For example,

```
<%= weblogic.servlet.security.Utils.encodeXSS(
javax.servlet.ServletRequest.getParameter("userInput"))%>
```

To secure an entire application, you must use the `encodeXSS()` method **each time** you return user-supplied data. While the previous example is an obvious location in which to use the `encodeXSS()` method, Table 10-3 describes other locations to consider using the `encodeXSS()` method.

Table 10-3 Code that Returns User-Supplied Data

Page Type	User-Supplied Data	Example
Error page	Erroneous input string, invalid URL, username	An error page that says "*`username`* is not permitted access."
Status page	Username, summary of input from previous pages	A summary page that asks a user to confirm input from previous pages.
Database display	Data presented from a database	A page that displays a list of database entries that have been previously entered by a user.

Using Sessions with JSP

Sessions in WebLogic JSP perform according to the JSP 2.0 specification. The following suggestions pertain to using sessions:

- Store small objects in sessions. For example, a session should not be used to store an EJB, but an EJB primary key instead. Store large amounts of data in a database. The session should hold only a simple string reference to the data.
- When you use sessions with dynamic reloading of servlets or JSPs, the objects stored in the servlet session must be serializable. Serialization is required because the servlet is reloaded in a new class loader, which results in an incompatibility between any classes loaded previously (from the old version of the servlet) and any classes loaded in the new class loader (for the new version of the servlet classes). This incompatibility causes the servlet to return `ClassCastException` errors.
- If session data *must* be of a user-defined type, the data class should be serializable. Furthermore, the session should store the serialized representation of the data object. Serialization should be compatible across versions of the data class.

Deploying Applets from JSP

Using the JSP provides a convenient way to include the Java Plug-in in a Web page, by generating HTML that contains the appropriate client browser tag. The Java Plug-in allows you to use a Java

Runtime Environment (JRE) supplied by Sun Microsystems instead of the JVM implemented by the client Web browser. This feature avoids incompatibility problems between your applets and specific types of Web browsers. The Java Plug-in is available from Sun at `http://java.sun.com/products/plugin/`.

Because the syntax used by Internet Explorer and Netscape is different, the servlet code generated from the `<jsp:plugin>` action dynamically senses the type of browser client and sends the appropriate `<OBJECT>` or `<EMBED>` tags in the HTML page.

The `<jsp:plugin>` tag uses many attributes similar to those of the `<APPLET>` tag, and some other attributes that allow you to configure the version of the Java Plug-in to be used. If the applet communicates with the server, the JVM running your applet code must be compatible with the JVM running WebLogic Server.

In the following example, the plug-in action is used to deploy an applet:

```
<jsp:plugin type="applet" code="examples.applets.PhoneBook1"
 codebase="/classes/" height="800" width="500"
 jreversion="2.0"
 nspluginurl=
 "http://java.sun.com/products/plugin/1.1.3/plugin-install.html"
 iepluginurl=
"http://java.sun.com/products/plugin/1.1.3/
  jinstall-113-win32.cab#Version=1,1,3,0" >

<jsp:params>
  <param name="weblogic_url" value="t3://localhost:7001">
  <param name="poolname" value="demoPool">
</jsp:params>

<jsp:fallback>
  <font color=#FF0000>Sorry, cannot run java applet!!</font>
</jsp:fallback>


</jsp:plugin>
```

The sample JSP syntax shown here instructs the browser to download the Java Plug-in version 1.3.1 (if it has not been downloaded previously), and run the applet identified by the `code` attribute from the location specified by `codebase`.

The `jreversion` attribute identifies the spec version of the Java Plug-in that the applet requires to operate. The Web browser attempts to use this version of the Java Plug-in. If the plug-in is not

already installed on the browser, the `nspluginurl` and `iepluginurl` attributes specify URLs where the Java Plug-in can be downloaded from the Sun Web site. Once the plug-in is installed on the Web browser, it is not downloaded again.

Because WebLogic Server uses the Java 1.3.x VM, you must specify the Java Plug-in version 1.3.x in the `<jsp:plugin>` tag. To specify the 1.3 JVM in the previous example code, replace the corresponding attribute values with the following:

```
jreversion="1.3"
nspluginurl=
"http://java.sun.com/products/plugin/1.3/plugin-install.html"
iepluginurl=
"http://java.sun.com/products/plugin/1.3/jinstall-131-win32.cab"
```

The other attributes of the plug-in action correspond with those of the `<APPLET>` tag. You specify applet parameters within a pair of `<params>` tags, nested within the `<jsp:plugin>` and `</jsp:plugin>` tags.

The `<jsp:fallback>` tags allow you to substitute HTML for browsers that are not supported by the `<jsp:plugin>` action. The HTML nested between the `<fallback>` and `</jsp:fallback>` tags is sent instead of the plug-in syntax.

Using the WebLogic JSP Compiler

The WebLogic JSP compiler is deprecated. BEA recommends that you use the WebLogic appc compiler, weblogic.appc, to compile EAR files, WAR files and EJBs.

For better compilation performance, the WebLogic JSP compiler transforms a JSP directly into a class file on the disk instead of first creating a java file on the disk and then compiling it into a class file. The java file only resides in memory.

To see the generated java file, turn on the -keepgenerated flag which dumps the in-memory java file to the disk.

Note: During JSP compilation, neither the command line flag (compilerclass) nor the descriptor element is invoked.

JSP Compiler Syntax

The JSP compiler works in much the same way that other WebLogic compilers work (including the RMI and EJB compilers). To start the JSP compiler, enter the following command.

```
$ java weblogic.jspc -options fileName
```

Replace *fileName* with the name of the JSP file that you want to compile. You can specify any *options* before or after the target *fileName*. The following example uses the -d option to compile myFile.jsp into the destination directory, weblogic/classes:

```
$ java weblogic.jspc -d /weblogic/classes myFile.jsp
```

Note: If you are precompiling JSPs that are part of a Web Application and that reference resources in the Web Application (such as a JSP tag library), you must use the -webapp flag to specify the location of the Web Application. The -webapp flag is described in the following listing of JSP compiler options.

JSP Compiler Options

Use any combination of the following options:

-classpath
: Add a list (separated by semi-colons on Windows NT/2000 platforms or colons on UNIX platforms) of directories that make up the desired CLASSPATH. Include directories containing any classes required by the JSP. For example (to be entered on one line):

```
$ java weblogic.jspc
   -classpath java/classes.zip;/weblogic/classes.zip
   myFile.JSP
```

-charsetMap
: Specifies mapping of IANA or unofficial charset names used in JSP contentType directives to java charset names. For example:
`-charsetMap x-sjis=Shift_JIS,x-big5=Big5`

  The most common mappings are built into the JSP compiler. Use this option only if a desired charset mapping is not recognized.

-commentary
: Causes the JSP compiler to include comments from the JSP in the generated HTML page. If this option is omitted, comments do not appear in the generated HTML page.

-compileAll
: Recursively compiles all JSPs in the current directory, or in the directory specified with the -webapp flag. (See the listing for -webapp in this list of options.). JSPs in subdirectories are also compiled.

-compileFlags
: Passes one or more command-line flags to the compiler. Enclose multiple flags in quotes, separated by a space. For example:
`java weblogic.jspc -compileFlags "-g -v" myFile.jsp`

`-compiler`
: Specifies the Java compiler to be used to compile the class file from the generated Java source code. The default compiler used is `javac`. The Java compiler program should be in your `PATH` unless you specify the absolute path to the compiler explicitly.

`-compilerclass`
: Runs a Java compiler as a Java class and not as a native executable.

`-d <dir>`
: Specifies the destination of the compiled output (that is, the class file). Use this option as a shortcut for placing the compiled classes in a directory that is already in your `CLASSPATH`.

`-depend`
: If a previously generated class file for a JSP has a more recent date stamp than the JSP source file, the JSP is not recompiled.

`-debug`
: Compile with debugging on.

`-deprecation`
: Warn about the use of deprecated methods in the generated Java source file when compiling the source file into a class file.

`-docroot directory`
: See `-webapp`.

`-encoding default|named character encoding`
: Valid arguments include (a) `default` which specifies using the default character encoding of your JDK, (b) a named character encoding, such as `8859_1`. If the `-encoding` flag is not specified, an array of bytes is used.

`-g`
: Instructs the Java compiler to include debugging information in the class file.

`-help`
: Displays a list of all the available flags for the JSP compiler.

`-J`
: Takes a list of options that are passed to your compiler.

`-k`
: When compiling multiple JSPs with a single command, the compiler continues compiling even if one or more of the JSPs failed to compile.

`-keepgenerated`
: Keeps the Java source code files that are created as an intermediary step in the compilation process. Normally these files are deleted after compilation.

`-noTryBlocks`
: 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.

`-nowarn`
: Turns off warning messages from the Java compiler.

`-noPrintNulls`
: Shows "null" in jsp expressions as "".

`-O`
: Compiles the generated Java source file with optimization turned on. This option overrides the `-g` flag.

`-package` *`packageName`*
: Sets the package name that is prepended to the package name of the generated Java HTTP servlet. Defaults to `jsp_servlet`.

`-superclass` *`classname`*
: Sets the classname of the superclass extended by the generated servlet. The named superclass must be a derivative of `HttpServlet` or `GenericServlet`.

`-verbose`
: Passes the `verbose` flag to the Java compiler specified with the `compiler` flag. See the compiler documentation for more information. The default is off.

`-verboseJavac`
: Prints messages generated by the designated JSP compiler.

`-version`
: Prints the version of the JSP compiler.

`-webapp` *`directory`*
: Name of a directory containing a Web Application in exploded directory format. If your JSP contains references to resources in a Web Application such as a JSP tag library or other Java classes, the JSP compiler will look for those resources in this directory. If you omit this flag when compiling a JSP that requires resources from a Web Application, the compilation will fail.

Precompiling JSPs

You can configure WebLogic Server to precompile your JSPs when a Web Application is deployed or re-deployed or when WebLogic Server starts up by setting the `precompile` parameter to true in the `<jsp-descriptor>` element of the `weblogic.xml` deployment descriptor. To avoid recompiling your JSPs each time the server restarts and when you target

additional servers, precompile them using weblogic.jspc and place them in the `WEB-INF/classes` folder and archive them in a .war file. Keeping your source files in a separate directory from the archived .war file will eliminate the possibility of errors caused by a JSP having a dependency on one of the class files.

Using the JSPClassServlet

Another way to prevent your JSPs from recompiling is to use the JSPClassServlet in place of JSPServlet and to place your precompiled JSPs into the WEB-INF/classes directory. This will remove any possibility of the JSPs being recompiled, as the server will not look at the source code. The server will not note any changes to the JSPs and recompile them if you choose this option. This option allows you to completely remove the JSP source code from your application after precompiling.

This is an example of how to add the JSPClassServlet to your Web Application's `web.xml` file.

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

  <servlet-mapping>
    <servlet-name>JSPClassServlet</servlet-name>
    <url-pattern>*.jsp</url-pattern>
  </servlet-mapping>
```

As when using virtual hosting, you must have physical directories that correspond to the mappings you create to allow your files to be found by the server.

CHAPTER 11

Filters

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

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.

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

Uses for Filters

Filters can be useful for the following functions:

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

Writing a Filter Class

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

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

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

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

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

Configuring Filters

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

Configuring a Filter

To configure a filter:

1. Open the `web.xml` deployment descriptor in a text editor or use the Administration Console. For more information, see "Web Application Developer Tools" on page 2-6. The `web.xml` file is located in the `WEB-INF` directory of your Web application.
2. Add a filter declaration. The `filter` element declares a filter, defines a name for the filter, and specifies the Java class that executes the filter. The `filter` element must directly follow the `context-param` element and directly precede the `listener` and `servlet` elements. For example:

```
<context-param>Param</context-param>

<filter>
  <icon>
    <small-icon>MySmallIcon.gif</small-icon>
    <large-icon>MyLargeIcon.gif</large-icon>
  </icon>
  <filter-name>myFilter</filter-name>
  <display-name>My Filter</display-name>
```

```
  <description>This is my filter</description>
  <filter-class>examples.myFilterClass</filter-class>
</filter>

<listener>Listener</listener>

<servlet>Servlet</servlet>
```

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

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

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

Your `Filter` class can read the initialization attributes using the `FilterConfig.getInitParameter()` or `FilterConfig.getInitParameters()` methods.

4. Add filter mappings. The `filter-mapping` element specifies which filter to execute based on a URL pattern or servlet name. The `filter-mapping` element must immediately follow the `filter` element(s).

   – To create a filter mapping using a URL pattern, specify the name of the filter and a URL pattern. URL pattern matching is performed according to the rules specified in the Servlet 2.4 Specification from Sun Microsystems at `http://java.sun.com/products/servlet/download.html#specs`, 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-mapping>
```

5. To create a chain of filters, specify multiple filter mappings. For more information, see "Configuring a Chain of Filters" on page 11-5.

Configuring a Chain of Filters

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

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

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

Filtering the Servlet Response Object

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

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

Additional Resources

- Servlet 2.4 Specification from Sun Microsystems at
  http://java.sun.com/products/servlet/download.html#specs
- J2EE API Reference (Javadocs) from Sun Microsystems 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 12

Using WebLogic JSP Form Validation Tags

The following sections describe how to use WebLogic JSP form validation tags:

Overview of WebLogic JSP Form Validation Tags

WebLogic JSP form validation tags provide a convenient way to validate the entries an end user makes to HTML form text fields generated by JSP pages. Using the WebLogic JSP form validation tags prevents unnecessary and repetitive coding of commonly used validation logic. The validation is performed by several custom JSP tags that are included with the WebLogic Server distribution. The tags can

- Verify that required fields have been filled in (`Required Field Validator` class).
- Validate the text in the field against a regular expression (`Regular Expression Validator` class).
- Compare two fields in the form (`Compare Validator` class).

- Perform custom validation by means of a Java class that you write (`Custom Validator` class).

WebLogic JSP form validation tags include:

- `<wl:summary>`
- `<wl:form>`
- `<wl:validator>`

When a validation tag determines that data in a field is not been input correctly, the page is re-displayed and the fields that need to be re-entered are flagged with text or an image to alert the end user. Once the form is correctly filled out, the end user's browser displays a new page specified by the validation tag.

Validation Tag Attribute Reference

This section describes the WebLogic form validation tags and their attributes. Note that the prefix used to reference the tag can be defined in the `taglib` directive on your JSP page. For clarity, the `wl` prefix is used to refer to the WebLogic form validation tags throughout this document.

<wl:summary>

`<wl:summary>` is the parent tag for validation. Place the opening `<wl:summary>` tag before any other element or HTML code in the JSP. Place the closing `</wl:summary>` tag anywhere *after* the closing `</wl:form>` tag(s).

`name`

(Optional) Name of a vector variable that holds all validation error messages generated by the `<wl:validator>` tags on the JSP page. If you do not define this attribute, the default value, `errorVector`, is used. The text of the error message is defined with the `errorMessage` attribute of the `<wl:validator>` tag.

To display the values in this vector, use the `<wl:errors/>` tag. To use the `<wl:errors/>` tag, place the tag on the page where you want the output to appear. For example:

```
<wl:errors color="red"/>
```

Alternately, you can use a scriptlet. For example:

```
<% if (errorVector.size() > 0) {
   for (int i=0; i < errorVector.size(); i++) {
   out.println((String)errorVector.elementAt(i));
   out.println("<br>");
```

```
   }
} %>
```

Where *errorVector* is the name of the vector assigned using the name attribute of the <wl:summary> tag.

The name attribute is required when using multiple forms on a page.

headerText
: A variable that contains text that can be displayed on the page. If you only want this text to appear when errors occur on the page, you can use a scriptlet to test for this condition. For example:

```
<% if(summary.size() >0 ) {
   out.println(headerText);
   }
%>
```

Where *summary* is the name of the vector assigned using the name attribute of the <wl:summary> tag.

redirectPage
: URL for the page that is displayed if the form validation does not return errors. This attribute is not required if you specify a URL in the action attribute of the <wl:form> tag.

Note: Do not set the redirectPage attribute to the same page containing the <wl:summary> tag—you will create an infinite loop causing a StackOverFlow exception.

<wl:form>

The <wl:form> tag is similar to the HTML <form> tag and defines an HTML form that can be validated using the WebLogic JSP form validation tags. You can define multiple forms on a single JSP by uniquely identifying each form using the name attribute.

method
: Enter GET or POST. Functions exactly as the method attribute of the HTML <form> tag.

action
: URL for the page that is displayed if the form validation does not return errors. The value of this attribute takes precedence over the value of the redirectPage attribute of the <wl:summary> tag and is useful if you have multiple forms on a single JSP page.

Note: Do not set the action attribute to the same page containing the <wl:form> tag—you will create an infinite loop causing a StackOverFlow exception.

`name`

Functions exactly as the `name` attribute of the HTML `<form>` tag. Identifies the form when multiple forms are used on the same page. The `name` attribute is also useful for JavaScript references to a form.

<wl:validator>

Use one or more `<wl:validator>` tags for each form field. If, for instance, you want to validate the input against a regular expression and also require that something be entered into the field you would use two `<wl:validator>` tags, one using the `RequiredFieldValidator` class and another using the `RegExpValidator` class. (You need to use both of these validators because blank values are evaluated by the Regular Expression Field Validator as valid.)

`errorMessage`

A string that is stored in the vector variable defined by the `name` attribute of the `<wl:summary>` tag.

`expression`

When using the `RegExpValidator` class, the regular expression to be evaluated.

If you are not using `RegExpValidator`, you can omit this attribute.

`fieldToValidate`

Name of the form field to be validated. The name of the field is defined with the `name` attribute of the HTML `<input>` tag.

`validatorClass`

The name of the Java class that executes the validation logic. Three classes are provided for your use. You can also create your own custom validator class. For more information, see "Using a Custom Validator Class" on page 8.

The available validation classes are:

`weblogicx.jsp.tags.validators.RequiredFieldValidator`

Validates that some text has been entered in the field.

`weblogicx.jsp.tags.validators.RegExpValidator`

Validates the text in the field using a standard regular expression.

Note: A blank value is evaluated as valid.

`weblogicx.jsp.tags.validators.CompareValidator`

Checks to see if two fields contain the same string. When using this class, set the `fieldToValidate` attribute to the two fields you want to compare. For example:

```
fieldToValidate="field_1,field_2"
```

Note: If both fields are blank, the comparison is evaluated as valid.

myPackage.myValidatorClass
: Specifies a custom validator class.

Using WebLogic JSP Form Validation Tags in a JSP

To use a validation tag in a JSP:

1. Write the JSP.

   a. Enter a `taglib` directive to reference the tag library containing the WebLogic JSP Form Validation Tags. For example:

      ```
      <%@ taglib uri="tagl" prefix="wl" %>
      ```

      Note that the prefix attribute defines the prefix used to reference all tags in your JSP page. Although you may set the prefix to any value you like, the tags referred to in this document use the `wl` prefix.

   b. Enter the `<wl:summary> ... </wl:summary>` tags.

      Place the opening `<wl:summary ...>` tag *before* any HTML code, JSP tag, scriptlet, or expression on the page.

      Place the closing `</wl:summary>` tag anywhere *after* the `</wl:form>` tag(s).

   c. Define an HTML form using the `<wl:form>` JSP tag that is included with the supplied tag library. For more information, see "<wl:form>" on page 3 and "Creating HTML Forms Using the <wl:form> Tag" on page 6. Be sure to close the form block with the `</wl:form>` tag. You can create multiple forms on a page if you uniquely define the `name` attribute of the `<wl:form>` tag for each form.

   d. Create the HTML form fields using the HTML `<input>` tag.

   e. Add `<wl:validator>` tags. For the syntax of the tags, see "<wl:validator>" on page 4. Place `<wl:validator>` tags on the page where you want the error message or image to appear. If you use multiple forms on the same page, place the `<wl:validator>` tag inside the `<wl:form>` block containing the form fields you want to validate.

      The following example shows a validation for a required field:

      ```
      <wl:form name="FirstForm" method="POST" action="thisJSP.jsp">

      <wl:validator
       errorMessage="Field_1 is required" expression=""
       fieldToValidate="field_1"
       validatorClass=
        "weblogicx.jsp.tags.validators.RequiredFieldValidator"
      ```

```
>
  <img src="images/warning.gif">
  <font color=red>Field 1 is a required field</font>
</wl:validator>

<p> <input type="text" name = "field_1">  </p>
<p> <input type="text" name = "field_2">  </p>
<p> <input type="submit" value="Submit FirstForm">  </p>
</wl:form>
```

If the user fails to enter a value in `field_1`, the page is redisplayed, showing a `warning.gif` image, followed by the text (in red) "`Field 1 is a required field,`" followed by the blank field for the user to re-enter the value.

2. Copy the `weblogic-vtags.jar` file from the `ext` directory of your WebLogic Server installation into the `WEB-INF/lib` directory of your Web Application. You may need to create this directory.

3. Configure your Web Application to use the tag library by adding a `taglib` element to the `web.xml` deployment descriptor for the Web Application. For example:

```
<taglib>
  <taglib-uri>tagl</taglib-uri>
  <taglib-location>
    /WEB-INF/lib/weblogic-vtags.jar
  </taglib-location>
</taglib>
```

Creating HTML Forms Using the <wl:form> Tag

This section contains information on creating HTML forms in your JSP page. You use the `<wl:form>` tag to create a single form or multiple forms on a page.

Defining a Single Form

Use the `<wl:form>` tag that is provided in the `weblogic-vtags.jar` tag library: For example:

```
<wl:form method="POST" action="nextPage.jsp">
<p> <input type="text" name ="field_1">  </p>
<p> <input type="text" name ="field_2">  </p>
<p> <input type="submit" value="Submit Form">  </p>
</wl:form>
```

For information on the syntax of this tag see "<wl:form>" on page 3.

Defining Multiple Forms

When using multiple forms on a page, use the `name` attribute to identify each form. For example:

```
<wl:form name="FirstForm" method="POST" action="thisJSP.jsp">
<p> <input type="text" name="field_1">  </p>
<p> <input type="text" name="field_2">  </p>
<p> <input type="submit" value="Submit FirstForm">  </p>
</wl:form>

<wl:form name="SecondForm" method="POST" action="thisJSP.jsp">
<p> <input type="text" name="field_1">  </p>
<p> <input type="text" name="field_2">  </p>
<p> <input type="submit" value="Submit SecondForm">  </p>
</wl:form>
```

Re-Displaying the Values in a Field When Validation Returns Errors

When the JSP page is re-displayed after the validator tag has found errors, it is useful to re-display the values that the user already entered, so that the user does not have to fill out the entire form again. Use the `value` attribute of the HTML `<input>` tag or use a tag library available from the Apache Jakarta Project. Both procedures are described next.

Re-Displaying a Value Using the <input> Tag

You can use the `javax.servlet.ServletRequest.getParameter()` method together with the `value` attribute of the HTML `<input>` tag to re-display the user's input when the page is re-displayed as a result of failed validation. For example:

```
<input type="text" name="field_1"
 value="<%= request.getParameter("field_1") %>" >
```

To prevent cross-site scripting security vulnerabilities, replace any HTML special characters in user-supplied data with HTML entity references. For more information, refer to "JSP Expression Language" on page 10-12.

Re-Displaying a Value Using the Apache Jakarta <input:text> Tag

You can also use a JSP tag library available free from the Apache Jakarta Project, which provides the `<input:text>` tag as a replacement for the HTML `<input>` tag. For example, the following HTML tag:

```
<input type="text" name="field_1">
```

could be entered using the Apache tag library as:

```
<input:text name="field_1">
```

For more information and documentation, download the Input Tag library, available at `http://jakarta.apache.org/taglibs/doc/input-doc/intro.html`.

To use the Apache tag library in your JSP:

1. Copy the `input.jar` file from the Input Tag Library distribution file into the `WEB-INF/lib` directory of your Web Application.
2. Add the following directive to your JSP:

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

3. Add the following entry to the `web.xml` deployment descriptor of your Web application:

   ```
   <taglib>
       <taglib-uri>input</taglib-uri>
       <taglib-location>/WEB-INF/lib/input.jar</taglib-location>
   </taglib>
   ```

Using a Custom Validator Class

To use your own validator class:

1. Write a Java class that extends the `weblogicx.jsp.tags.validators.CustomizableAdapter` abstract class. For more information, see "Extending the CustomizableAdapter Class" on page 9.
2. Implement the `validate()` method. In this method:
   a. Look up the value of the field you are validating from the `ServletRequest` object. For example:

      ```
      String val = req.getParameter("field_1");
      ```

   b. Return a value of `true` if the field meets the validation criteria.
3. Compile the validator class and place the compiled `.class` file in the `WEB-INF/classes` directory of your Web application.
4. Use your validator class in a `<wl:validator>` tag by specifying the class name in the `validatorClass` attribute. For example:

```
<wl:validator errorMessage="This field is required"
fieldToValidate="field_1" validatorClass="mypackage.myCustomValidator">
```

Extending the CustomizableAdapter Class

The `CustomizableAdapter` class is an abstract class that implements the `Customizable` interface and provides the following helper methods:

`getFieldToValidate()`
: Returns the name of the field being validated (defined by the `fieldToValidate` attribute in the `<wl:validator>` tag)

`getErrorMessage()`
: Returns the text of the error message defined with the `errorMessage` attribute in the `<wl:validator>` tag.

`getExpression()`
: Returns the text of the `expression` attribute defined in the `<wl:validator>` tag.

Instead of extending the `CustomizableAdapter` class, you can implement the `Customizable` interface. For more information, see the Javadocs for weblogicx.jsp.tags.validators.Customizable.

Sample User-Written Validator Class

Listing 12-1 Example of a User-written Validator Class

```
import weblogicx.jsp.tags.validators.CustomizableAdapter;

public class myCustomValidator extends CustomizableAdapter{

    public myCustomValidator(){
       super();
    }

    public boolean validate(javax.servlet.ServletRequest req)
       throws Exception {
       String val = req.getParameter(getFieldToValidate());
        // perform some validation logic
        // if the validation is successful, return true,
         // otherwise return false
```

```
        if (true) {
            return true;
        }
        return false;
    }

}
```

Sample JSP with Validator Tags

This sample code shows the basic structure of a JSP that uses the WebLogic JSP form validation tags. A complete functioning code example is also available if you installed the examples with your WebLogic Server installation. Instructions for running the example are available at `samples/examples/jsp/tagext/form_validation/package.html`, in your WebLogic Server installation.

Listing 12-2 JSP with WebLogic JSP Form Validation Tags

```
<%@ taglib uri="tagl" prefix="wl" %>
<%@ taglib uri="input" prefix="input" %>


<wl:summary
name="summary"
headerText="<font color=red>Some fields have not been filled out
correctly.</font>"
redirectPage="successPage.jsp"
>

<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">
```

```
<% if(summary.size() >0 )   {
    out.println("<h3>" + headerText + "</h3>");
} %>


<% if (summary.size() > 0) {
       out.println("<H2>Error Summary:</h2>");
       for (int i=0; i < summary.size(); i++) {
       out.println((String)summary.elementAt(i));
       out.println("<br>");
       }
} %>


<wl:form method="GET" action="successPage.jsp">

 User Name: <input:text name="username"/>
 <wl:validator
   fieldToValidate="username"
   validatorClass="weblogicx.jsp.tags.validators.RequiredFieldValidator"
   errorMessage="User name is a required field!"
 >
   <img src=images/warning.gif> This is a required field!
 </wl:validator>

<p>

 Password: <input type="password" name="password">
 <wl:validator
   fieldToValidate="password"
   validatorClass="weblogicx.jsp.tags.validators.RequiredFieldValidator"
   errorMessage="Password is a required field!"
 >
   <img src=images/warning.gif> This is a required field!
 </wl:validator>
```

```
 <p>

 Re-enter Password: <input type="password" name="password2">
 <wl:validator
   fieldToValidate="password,password2"
   validatorClass="weblogicx.jsp.tags.validators.CompareValidator"
   errorMessage="Passwords don't match"
 >
   <img src=images/warning.gif> Passwords don't match.
 </wl:validator>

 <p>

 <input type="submit" value="Submit Form">  </p>

</wl:form>

</wl:summary>


</body>
</html>
```

CHAPTER 13

Using Custom WebLogic JSP Tags (cache, process, repeat)

The following sections describe the use of three custom JSP tags—`cache`, `repeat`, and `process`—provided with the WebLogic Server distribution:

Overview of WebLogic Custom JSP Tags

BEA provides three specialized JSP tags that you can use in your JSP pages: `cache`, `repeat`, and `process`. These tags are packaged in a tag library jar file called `weblogic-tags.jar`. This jar file contains classes for the tags and a tag library descriptor (TLD). To use these tags, you copy

this jar file to the Web application that contains your JSPs and reference the tag library in your JSP.

Using the WebLogic Custom Tags in a Web Application

Using the WebLogic custom tags requires that you include them within a Web application.

To use these tags in your JSP:

1. Copy the `weblogic-tags.jar` file from the `ext` directory of your WebLogic Server installation to the `WEB-INF/lib` directory of the Web application containing the JSPs that will use the WebLogic Custom Tags.
2. Reference this tag library descriptor in the `<taglib>` element of the J2EE standard Web application deployment descriptor, `web.xml`. For example:

```
<taglib>
  <taglib-uri>weblogic-tags.tld</taglib-uri>
  <taglib-location>
     /WEB-INF/lib/weblogic-tags.jar
  </taglib-location>
</taglib>
```

3. Reference the tag library in your JSP with the `taglib` directive. For example:

```
<%@ taglib uri="weblogic-tags.tld" prefix="wl" %>
```

Cache Tag

The cache tag enables caching the work that is done within the body of the tag. It supports both output (transform) data and input (calculated) data. Output caching refers to the content generated by the code within the tag. Input caching refers to the values to which variables are set by the code within the tag. Output caching is useful when the final form of the content is the important thing to cache. Input caching is important when the view of the data can vary independently of the data calculated within the tag.

If one client is already recalculating the contents of a cache and another client requests the same content it does not wait for the completion of the recalculation, instead it shows whatever information is already in the cache. This is to make sure that the web site does not come to a halt for all your users because a cache is being recalculated. Additionally, the async attribute means that no one, not even the user that initiates the cache recalculation waits.

Two versions of the cache tag are available. Version 2 has additional scopes available.

Caches are stored using soft references to prevent the caching system from using too much system memory. Unfortunately, due to incompatibilities with the HotSpot VM and the Classic VM, soft references are not used when WebLogic Server is running within the HotSpot VM.

Refreshing a Cache

You can force the refresh of a cache by setting the `_cache_refresh` object to `true` in the scope that you want affected. For example, to refresh a cache at session scope, specify the following:

```
<% request.setAttribute("_cache_refresh", "true"); %>
```

If you want all caches to be refreshed, set the cache to the `application` `scope`. If you want all the caches for a user to be refreshed, set it in the `session` scope. If you want all the caches in the current request to be refreshed, set the `_cache_refresh` object either as a parameter or in the request.

The `<wl:cache>` tag specifies content that must be updated each time it is displayed. The statements between the `<wl:cache>` and `</wl:cache>` tags are only executed if the cache has expired or if any of the values of the key attributes (see the Cache Tag Attributes table) have changed.

Flushing a Cache

Flushing a cache forces the cached values to be erased; the next time the cache is accessed, the values are recalculated. To flush a cache, set its `flush` attribute to `true`. The cache must be named using the `name` attribute. If the cache has the `size` attribute set, all values are flushed. If the cache sets the `key` attribute but not the `size` attribute, you can flush a specific cache by specifying its `key` along with any other attributes required to uniquely identify the cache (such as `scope` or `vars`).

For example:

1. Define the cache.

   ```
   <wl:cache name="dbtable" key="parameter.tablename"
   scope="application">
   // read the table and output it to the page
   </wl:cache>
   ```

2. Update the cached table data.

3. Flush the cache using the `flush` attribute in an empty tag (an empty tag ends with `/` and does not use a closing tag). For example

```
<wl:cache name="dbtable" key="parameter.tablename" scope="application"
flush="true"/>
```

Table 13-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
timeout	no	-1	Cache timeout property. The amount of time, in seconds, after which the statements within the cache tag are refreshed. This is not proactive; the value is refreshed only if it is requested. If you prefer to use a unit of time other than seconds, you can specify an alternate unit by postfixing the value with desired unit: ms = milliseconds s = seconds (default) m = minutes h = hours d = days
scope	no	application	Specifies the scope in which the data is cached. Valid scopes include: • parameter, (versions 1,2)requests the HTTP servlet request parameters • page, (versions 1,2)requests the JSP page context attributes (This scope does not exist for the cache filter.) • request, (versions 1,2)requests the servlet request attributes. Request attributes are valid for the entire request, including any forwarded or included pages. • cookie, (version 2)requests the cookie values found in the request. If there are multiple cookies with the same name, this request returns only the first value. • requestHeader, (version 2)requests the values from the request Headers. If there are multiple Headers with the same name, only the value of the first is returned.

Table 13-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
scope (cont.)			• responseHeader, (version 2)requests the values from the response Headers. If there are multiple Headers with the same name, only the value of the first is returned. If you set a response header, all response headers are replaced with the value you have set. This scope should not be used for storing content. • session, (versions 1,2)requests the values from the session attributes of the current user. If there is no session then one will not be created by accessing the scope. The caches can become very large if you are caching content. • application, (versions 1,2)requests the values found in the servlet context attributes. • cluster, (versions 1,2)requests the values from the application scope, and when written to replicates the information across the cluster. Most caches will be either session or application scope.

Table 13-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
`key`	`no`	`--`	Specifies additional values to be used when evaluating whether to cache the values contained within the tags. Typically a given cache is identified by the cache name that you configured in web.xml. If that is not specified the request uri is used as a cache name. Using keys you can specify additional values to identify a tag. For example, if you want to separate out the cache for a given end user, then in addition to the cache name you can specify the keys as the userid, values for which you want to pick it up from the request parameter scope (query param/post params) plus perhaps a client ip. So you will specify your keys as: "`parameter.userid,parameter.clientip`" Here "parameter" is the scope (request parameter scope) and "userid"/"clientip" are the parameters/attributes. This means the primary key for the cache becomes the cache name (request uri in this case) + value of userid request param + value of clientip request param. The list of keys is comma-separated. The value of this attribute is the name of the variable whose value you wish to use as a key into the cache. You can additionally specify a scope by prepending the name of the scope to the name. For example: `parameter.key | page.key | request.key | application.key | session.key` It defaults to searching through the scopes in the order shown in the preceding list. Each named key is available in the cache tag as a scripting variable. A list of keys is comma-separated.
`async`	`no`	`false`	If the `async` parameter is set to `true`, the cache will be updated asynchronously, if possible. The user that initiates the cache hit sees the old data.

Table 13-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
`name`	`no`	`--`	A unique name for the cache that allows caches to be shared across multiple JSP pages. This same buffer is used to store the data for all pages using the named cache. This attribute is useful for textually included pages that need cache sharing. If this attribute is not set, a unique name is chosen for the cache. We recommended that you avoid manually calculating the name of the tag; the `key` functionality can be used equivalently in all cases. The name is calculated as `weblogic.jsp.tags.CacheTag.` plus the URI plus a generated number representing the tag in the page you are caching. If different URIs reach the same JSP page, the caches are not shared in the default case. Use named caches in this case. System named caches can not be flushed and refreshed automatically.
`size`	`no`	`-1` (unlimited)	For caches that use keys, the number of entries allowed. The default is an unlimited cache of keys. With a limited number of keys the tag uses a *least-used system* to order the cache. Changing the value of the size attribute of a cache that has already been used does not change the size of that cache.

Table 13-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
vars	no	--	In addition to caching the transformed output of the cache, you can also cache calculated values within the block. These variables are specified exactly the same way as the cache keys. This type of caching is called *Input caching*. Variables are used to do input caching. When the cache is retrieved the variables are restored to the scope you specified. For example, for retrieving results from a database you used var1 from request parameter and var2 from session. When the cache is created the value of these variables are stored with the cache. The next time the cache is accessed these values are restored so you will be able to access them from their respective scopes. For example, var1 will be available from request and var2 from session.
flush	no	none	When set to true, the cache is flushed. This attribute must be set in an empty tag (ends with /).

Additional properties of the cache system for version 2

- Each cache also has additional arbitrary attributes associated with it that the end user can manipulate and expect to be populated when the cache is retrieved.
- Cache listeners can be registered by putting an object that implements *weblogicx.cache.CacheListener* in a *java.util.List* that is present in any scope in the cache system under the "weblogicx.cache.CacheListener" key. If there is a *List* present in the scope, add your listener to the end.

The following examples show how you can use the `<wl:cache>` tag.

Listing 13-1 Examples of Using the cache Tag

```
<wl:cache>
<!--the content between these tags will only be
 refreshed on server restart-->
</wl:cache>
```

```
<wl:cache key="request.ticker" timeout="1m">
<!--get stock quote for whatever is in the request parameter ticker
 and display it, only update it every minute-->
</wl:cache>

<!--incoming parameter value isbn is the number used to lookup the
 book in the database-->
<wl:cache key="parameter.isbn" timeout="1d" size="100">
<!--retrieve the book from the database and display
the information -- the tag will cache the top 100
most accessed book descriptions-->
</wl:cache>

<wl:cache timeout="15m" async="true">
<!--get the new headlines from the database every 15 minutes and
 display them-->
<!--do not let anyone see the pause while they are retrieved-->
</wl:cache>
```

Process Tag

Use the `<wl:process>` tag for query parameter-based flow control. By using a combination of the tag's four attributes, you can selectively execute the statements between the `<wl:process>` and `</wl:process>` tags. The process tag may also be used to declaratively process the results of form submissions. By specifying conditions based on the values of request parameters you can include or not include JSP syntax on your page.

Table 13-2 Process Tag Attributes

Tag Attribute	Required	Description
name	no	Name of a query parameter.
notname	no	Name of a query parameter.

Table 13-2 Process Tag Attributes

Tag Attribute	Required	Description
value	no	Value of a query parameter.
notvalue	no	Value of a query parameter.

The following examples show how you can use the `<wl:process>` tag:

Listing 13-2 Examples of Using the process tag:

```
<wl:process notname="update">
<wl:process notname="delete">
<!--Only show this if there is no update or delete parameter-->
<form action="<%= request.getRequestURI() %>">
  <input type="text" name="name"/>
  <input type="submit" name="update" value="Update"/>
  <input type="submit" name="delete" value="Delete"/>
</form>
</wl:process>
</wl:process>
```

```
<wl:process name="update">
<!-- do the update -->
</wl:process>


<wl:process name="delete">
<!--do the delete-->
</wl:process>
```

```
<wl:process name="lastBookRead" value="A Man in Full">
<!--this section of code will be executed if lastBookRead exists
 and the value of lastBookRead is "A Man in Full"-->
</wl:process>
```

Repeat Tag

Use the `<wl:repeat>` tag to iterate over many different types of sets, including Enumerations, Iterators, Collections, Arrays of Objects, Vectors, ResultSets, ResultSetMetaData, and the keys of a Hashtable. You can also just loop a certain number of times by using the `count` attribute. Use the set attribute to specify the type of Java objects.

Table 13-3 Repeat Tag Attributes

Tag Attribute	Required	Type	Description
set	No	Object	The set of objects that includes: • Enumerations • Iterators • Collections • Arrays • Vectors • Result Sets • Result Set MetaData • Hashtable keys
count	No	Int	Iterate over first *`count`* entries in the set.
id	No	String	Variable used to store current object being iterated over.
type	No	String	Type of object that results from iterating over the set you passed in. Defaults to `Object`. This type must be fully qualified.

The following example shows how you can use the `<wl:repeat>` tag.

Listing 13-3 Examples of Using the repeat Tag

```
<wl:repeat id="name" set="<%= new String[] { "sam", "fred", "ed" } %>">
  <%= name %>
</wl:repeat>

<% Vector v = new Vector();%>
```

```
<!--add to the vector-->

<wl:repeat id="item" set="<%= v.elements() %>">
<!--print each element-->
</wl:repeat>
```

CHAPTER 14

Using the WebLogic EJB to JSP Integration Tool

The following sections describe how to use the WebLogic EJB-to-JSP integration tool to create JSP tag libraries that you can use to invoke EJBs in a JavaServer Page (JSP). This document assumes at least some familiarity with both EJB and JSP.

Overview of the WebLogic EJB-to-JSP Integration Tool

Given an EJB jar file, the WebLogic EJB-to-JSP integration tool will generate a JSP tag extension library whose tags are customized for calling the EJB(s) of that jar file. From the perspective of a client, an EJB is described by its remote interface. For example:

```
public interface Trader extends javax.ejb.EJBObject {
  public TradeResult buy(String stockSymbol, int shares);
  public TradeResult sell(String stockSymbol, int shares);
}
```

For Web Applications that call EJBs, the typical model is to invoke the EJB using Java code from within a JSP scriptlet (<% ... %>). The results of the EJB call are then formatted as HTML and presented to the Web client. This approach is both tedious and error-prone. The Java code required to invoke an EJB is lengthy, even in the simplest of cases, and is typically not within the skill set of most Web designers responsible for HTML presentation.

The EJB-to-JSP tool simplifies the EJB invocation process by removing the need for java code. Instead, you invoke the EJB is invoked using a JSP tag library that is custom generated for that EJB. For example, the methods of the Trader bean above would be invoked in a JSP like this:

```
<%@ taglib uri="/WEB-INF/trader-tags.tld" prefix="trade" %>
<b>invoking trade: </b><br>

<trade:buy stockSymbol="BEAS" shares="100"/>

<trade:sell stockSymbol="MSFT" shares="200"/>
```

The resulting JSP page is cleaner and more intuitive. A tag is (optionally) generated for each method on the EJB. The tags take attributes that are translated into the parameters for the corresponding EJB method call. The tedious machinery of invoking the EJB is hidden, encapsulated inside the handler code of the generated tag library. The generated tag libraries support stateless and stateful session beans, and entity beans. The tag usage scenarios for each of these cases are slightly different, and are described below.

Basic Operation

You can run the WebLogic EJB-to-JSP integration tool in command-line mode using the following command:

```
java weblogic.servlet.ejb2jsp.Main
```

or graphical mode. For all but the simplest EJBs, the graphical tool is preferable.

Invoke the graphical tool as follows:

```
java weblogic.servlet.ejb2jsp.gui.Main
```

Initially, no ejb2jsp project is loaded by the Web application. Create a new project by selecting the **File -> New** menu item, browsing in the file chooser to an EJB jar file, and selecting it. Once initialized, you can modify, save, and reload ejb2jsp projects for future modification.

The composition of the generated tag library is simple: for each method, of each EJB, in the jar file, a JSP tag is generated, with the same name as the method. Each tag expects as many attributes as the corresponding method has parameters.

Interface Source Files

When a new EJB jar is loaded, the tool also tries to find the Java source files for the home and remote interfaces of your EJB(s). The reason is that, although the tool can generate tags only by introspecting the EJB classes, it cannot assign meaningful attribute names to the tags whose corresponding EJB methods take parameters. In the **Trader** example in “Overview of the WebLogic EJB-to-JSP Integration Tool” on page 1, when the EJB jar is loaded, the tool tries to find a source file called **Trader.java**. This file is then parsed and detects that the **buy()** method takes parameters called **stockSymbol** and **shares**. The corresponding JSP tag will then have appropriately named attributes that correspond to the parameters of the **buy()** method.

When a new EJB jar is loaded, the tool operates on the premise that the source directory is the same directory where the EJB jar is located. If that is not the case, the error is not fatal. After the new project is loaded, under the **Project Build Options** panel, you can adjust the **EJB Source Path** element to reflect the correct directory. You can then select the **File -> Resolve Attributes** menu to re-run the resolve process.

When looking for java source files corresponding to an interface class, the tool searches in both the directory specified, and in a sub-directory implied by the interface's java package. For example, for **my.ejb.Trader**, if the directory given is **C:/src**, the tool will look for both **C:/src/Trader.java** and **C:/src/my/ejb/Trader.java**.

Access to the source files is not strictly necessary. You can always modify attribute names for each tag in a project by using the tool. However, parsing the source files of the EJB's public interface was developed as the quickest way to assign meaningful attribute names.

Build Options Panel

Use this panel to set all parameters related to the local file system that are needed to build the project. Specify the Java compiler, the Java package of the generated JSP tag handlers, and whether to keep the generated Java code after a project build, which can be useful for debugging.

You can also use this panel to specify the type of tag library output you want. For use in a J2EE web application, a tag library should be packaged one of two ways: as separate class files and a Tag Library Descriptor (.tld) file, or as a single taglib jar file. Either output type is chosen with the **Output Type** pull-down. For development and testing purposes, **DIRECTORY** output is recommended, because a Web Application in WebLogic Server must be re-deployed before a jar file can be overwritten.

For either **DIRECTORY** or **JAR**, the output locations must be chosen appropriately so that the tag library will be found by a web application. For example, if you wish to use the tag library in a web application rooted in directory **C:/mywebapp**, then the **DIRECTORY classes** field should be specified as:

```
C:/mywebapp/WEB-INF/classes
```

and the **DIRECTORY .tld File** field should be something like:

```
C:/mywebapp/WEB-INF/trader-ejb.tld
```

The **Source Path**, described earlier, is edited in the **Build Options** panel as well. The **Extra Classpath** field can be used if your tag library depends on other classes not in the core WebLogic Server or J2EE API. Typically, nothing will need to be added to this field.

Troubleshooting

Sometimes, a project fails to build because of errors or conflicts. This section describes the reasons for those errors, and how they may be resolved.

- **Missing build information** One of the necessary fields in the **Build Options** panel is unspecified, like the java compiler, the code package name, or a directory where the output can be saved. The missing field(s) must be filled in before the build can succeed.
- **Duplicate tag names** When an EJB jar is loaded, the tool records a tag for each method on the EJB, and the tag name is the same as the method name. If the EJB has overloaded methods (methods with the same name but different signatures), the tag names conflict. Resolve the conflict by renaming one of the tags or by disabling one of the tags. To rename a tag, navigate to the tag in question using the tree hierarchy in the left window of the tool. In the tag panel that appears in the right window, modify the **Tag Name** field. To disable a tag, navigate to the tag in question using the tree hierarchy in the left window of the tool. In the tag panel that appears in the right window, deselect the **Generate Tag** box. For EJB jars that contain multiple EJBs, you can disable tags for an entire bean may as well.
- **Meaningless attribute names arg0, arg1...** This error occurs when reasonable attribute names for a tag could not be inferred from the EJB's interface source files. To fix this error,

navigate to the tag in question in the project hierarchy tree. Select each of the attribute tree leaves below the tag, in order. For each attribute, assign a reasonable name to the **Attribute Name** field, in the panel that appears on the right side of the tool.

- **Duplicate attribute names** This occurs when a single tag expecting multiple attributes has two attributes with the same name. Navigate to the attribute(s) in question, and rename attributes so that they are all unique for the tag.

Using EJB Tags on a JSP Page

Using the generated EJB tags on a JSP page is simply a matter of declaring the tag library on the page, and then invoking the tags like any other tag extension:

```
<% taglib uri="/WEB-INF/trader-ejb.tld"
 prefix="trade" %>
<trade:buy stockSymbol="XYZ" shares="100"/>
```

For EJB methods that have a non-void return type, a special, optional tag attribute "_return", is built-in. When present, the value returned from the method is made available on the page for further processing:

```
<% taglib uri="/WEB-INF/trader-ejb.tld"
  prefix="trade" %>
<trade:buy stockSymbol="XYZ"
 shares="100" _return="tr"/>
<% out.println("trade result: " + tr.getShares()); %>
```

For methods that return a primitive numeric type, the return variable is a Java object appropriate for that type (for example, "int" -> java.lang.Integer, etc).

EJB Home Methods

EJB 2.0 allows for methods on the EJB home interface that are neither **create()** or **find()** methods. Tags are generated for these home methods as well. To avoid confusion, the tool prepends "**home-**" to the tags for each method on an EJB's home, when a new project is loaded. These methods may be renamed, if desired.

Stateful Session and Entity Beans

Typical usage of a “stateful” bean is to acquire an instance of the bean from the bean's Home interface, and then to invoke multiple methods on a single bean instance. This programming

model is preserved in the generated tag library as well. Method tags for stateful EJB methods are required to be inside a tag for the EJB home interface that corresponds to a **find()** or **create()** on the home. All EJB method tags contained within the find/create tag operate on the bean instance found or created by the enclosing tag. If a method tag for a stateful bean is not enclosed by a find/create tag for its home, a run-time exception occurs. For example, given the following EJB:

```
public interface AccountHome extends EJBHome {

  public Account create(String accountId, double initialBalance);
  public Account findByPrimaryKey(String accountID);
    /* find all accounts with balance above some threshold */
  public Collection findBigAccounts(double threshold);
}

public interface Account extends EJBObject {
  public String getAccountID();
  public double deposit(double amount);
  public double withdraw(double amount);
  public double balance();
}
```

Correct tag usage might be as follows:

```
<% taglib uri="/WEB-INF/account-ejb.tld" prefix="acct" %>
<acct:home-create accountId="103"
  initialBalance="450.0" _return="newAcct">
 <acct:deposit amount="20"/>
 <acct:balance _return="bal"/>
 Your new account balance is: <%= bal %>
</acct:home-create>
```

If the "_return" attribute is specified for a find/create tag, a page variable will be created that refers to the found/created EJB instance. Entity beans finder methods may also return a collection of EJB instances. Home tags that invoke methods returning a collection of beans will iterate (repeat) over their tag body, for as many beans as are returned in the collection. If "_return" is specified, it is set to the current bean in the iteration:

```
<b>Accounts above $500:</b>
<ul>
<acct:home-findBigAccounts threshold="500" _return="acct">
<li>Account <%= acct.getAccountID() %>
```

```
     has balance $<%= acct.balance() %>
</acct:home-findBigAccounts>
</ul>
```

The preceding example will display an HTML list of all Account beans whose balance is over $500.

Default Attributes

By default, the tag for each method requires that all of its attributes (method parameters) be set on each tag instance. However, the tool will also allow "default" method parameters to be specified, in case they are not given in the JSP tag. You can specify default attributes/parameters in the **Attribute** window of the EJB-to-JSP tool. The parameter default can come from an simple **EXPRESSION**, or if more complex processing is required, a default **METHOD** body may be written. For example, in the Trader example in “Overview of the WebLogic EJB-to-JSP Integration Tool” on page 1, suppose you want the “buy” tag to operate on stock symbol “XYZ” if none is specified. In the Attribute panel for the “stockSymbol” attribute of the “buy” tag, you set the “Default Attribute Value” field to **EXPRESSION**, and enter “XYZ” (quotes included!) in the **Default Expression** field. The buy tag then acts as if the stockSymbol="XYZ" attribute were present, unless some other value is specified.

Or if you want the shares attribute of the "buy" tag to be a random number between 0-100, we would set "Default Attribute Value" to **METHOD**, and in the **Default Method Body** area, you write the body of a Java method that returns int (the expected type for the "shares" attribute of the "buy" method):

```
long seed = System.currentTimeMillis();
java.util.Random rand = new java.util.Random(seed);
int ret = rand.nextInt();
/* ensure that it is positive...*/
ret = Math.abs(ret);
/* and < 100 */
return ret % 100;
```

Because your default method bodies appear within a JSP tag handler, your code has access to the **pageContext** variable. From the JSP PageContext, you can gain access to the current HttpServletRequest or HttpSession, and use session data or request parameters to generate default method parameters. For example, to pull the "shares" parameter for the "buy" method out of a ServletRequest parameter, you could write the following code:

```
HttpServletRequest req =
 (HttpServletRequest)pageContext.getRequest();
String s = req.getParameter("shares");
if (s == null) {
  /* webapp error handler will redirect to error page
   * for this exception
   */
  throw new BadTradeException("no #shares specified");
}
int ret = -1;
try {
  ret = Integer.parseInt(s);
} catch (NumberFormatException e) {
  throw new BadTradeException("bad #shares: " + s);
}
if (ret <= 0)
  throw new BadTradeException("bad #shares: " + ret);
return ret;
```

The generated default methods are assumed to throw exceptions. Any exceptions raised during processing will be handled by the JSP's errorPage, or else by the registered exception-handling pages of the Web Application.

APPENDIX A

web.xml Deployment Descriptor Elements

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

web.xml Namespace Declaration and Schema Location

The correct text for the namespace declaration and schema location for the web.xml file is as follows.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">
```

To view the schema for `web.xml`, go to http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd.

icon

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

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

Element	Required/ Optional	Description
`<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 contains the declaration of a Web application's servlet context initialization parameters. The following table describes the reserved context parameters used by the Web application container, which have been deprecated and have replacements in weblogic.xml.

Deprecated Parameter	Description	Replacement Element in weblogic.xml
`weblogic.httpd.inputCharset`	Defines code set behavior for non-unicode operations.	`input-charset` (defined within charset-param) in weblogic.xml. See "input-charset" on page B-22.
`weblogic.httpd.servlet.reloadCheckSecs`	Define how often WebLogic Server checks whether a servlet has been modified, and if so, reloads it. A value of -1 is never reload, 0 is always reload. The default is set to 1 second.	`servlet-reload-check-secs` (defined within `container-descriptor`) in `weblogic.xml`.See "container-descriptor" on page B-17.

Deprecated Parameter	Description	Replacement Element in weblogic.xml
`weblogic.httpd.servlet.classpath`	When this values has been set, the container appends this path to the Web application classpath. This is not a recommended method and is supported only for backward compatibility.	No replacement. Use other means such as manifest classpath or `WEB-INF/lib` or `WEB-INF/classes` or virtual directories.
`weblogic.httpd.defaultServlet`	Sets the default servlet for the Web application. This is not a recommended method and is supported only for backward compatibility.	No replacement. Instead use the `servlet` and `servlet-mapping` elements in `web.xml` to define a default servlet. The URL pattern for `default-servlet` should be "/". See "servlet-mapping" on page A-11. For additional examples of servlet mapping, see "Servlet Mapping" on page 4-2.

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

Element	Required/ Optional	Description
`weblogic.httpd. clientCertProxy`	optional	This attribute specifies that certifications from clients of the Web application are provided in the special `WL-Proxy-Client-Cert` header sent by a proxy plug-in or `HttpClusterServlet`. This setting is useful if user authentication is performed on a proxy server—setting `clientCertProxy` causes the proxy server to pass on the certs to the cluster in a special header, `WL-Proxy-Client-Cert`. A `WL-Proxy-Client-Cert` header could be provided by any client with access to WebLogic Server. WebLogic Server takes the certificate information from that header, trusting that is came from a secure source (the plug-in) and uses that information to authenticate the user. For this reason, if you set `clientCertProxy`, use a connection filter to ensure that WebLogic Server accepts connections only from the machine on which the plug-in is running. In addition to setting this attribute for an individual Web application, you can define this attribute: • For all web applications hosted by a server instance, on the Server-->Configuration-->General page in the Administration Console • For all web applications hosted by server instances in a cluster, on the Cluster-->Configuration-->General page.

filter

The `filter` element defines a filter class and its initialization attributes. For more information on filters, see "Configuring Filters" on page 11-3.

The following table describes the elements you can define within a `filter` 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 attribute of the filter. Use a separate set of `<init-param>` tags for each attribute.

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.

`<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 Class" on page 9-4.

servlet

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

If a `jsp-file` is specified and the `<load-on-startup>` element is present, then the JSP is precompiled and loaded when WebLogic Server starts.

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

Element	Required/ Optional	Description
`<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 attribute of the servlet. Use a separate set of `<init-param>` tags for each attribute.
`<load-on-startup>`	Optional	WebLogic Server initializes this servlet when WebLogic Server starts up. The optional content of this element must be a positive integer indicating the order in which the servlet should be loaded. Lower integers are loaded before higher integers. If no value is specified, or if the value specified is not a positive integer, WebLogic Server can load the servlet in any order during application startup.

Element	Required/ Optional	Description
`<run-as>`	Optional	Specifies the run-as identity to be used for the execution of the Web application. It contains an optional description and the name of a security role.
`<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-9.

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-9.

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

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

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

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

security-role-ref

This is an element within the "servlet" on page A-9.

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

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

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

servlet-mapping

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 2.3 Specification.
		For additional examples of servlet mapping, see "Servlet Mapping" on page 4-2.

session-config

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

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

Element	Required/ Optional	Description
`<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` attribute 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: 60 Maximum value: Integer.MAX_VALUE ÷ 60 Special values: • -1 = Sessions do not timeout. The value set in `<session-descriptor>` element of `weblogic.xml` is ignored. For more information, see "session-descriptor" on page B-7.

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 Files" on page 5-4.

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 5-5.

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`.

jsp-config

The `jsp-config` element is used to provide global configuration information for the JSP files in a Web application. It has two sub-elements, `taglib` and `jsp-property-group`.

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

Element	Required/ Optional	Description
`<taglib>`	Optional	Provides information on a tag library that is used by a JSP page within the Web application.
`<jsp-property-group>`	Optional	Used to group a number of files so they can be given global property information. All files so described are deemed to be JSP files.

taglib

This is an element within the "jsp-config" on page A-14.

The required `taglib` element provides information on a tag library that is used by a JSP page within the Web application.

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>`	Optional	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>`	Optional	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.

jsp-property-group

This is an element within the "jsp-config" on page A-14.

The required `jsp-property-group` element is used to group a number of files so they can be given global property information. All files so described are deemed to be JSP files.

The following table describes the elements you can define within a `jsp-property-group` element.

Element	Required/ Optional	Description
`<el-ignored>`	Optional	Controls whether EL is ignored. By default, the EL evaluation is enabled for Web Applications using a Servlet 2.4 or greater `web.xml`, and disabled otherwise.
`<scripting-invalid>`	Optional	Controls whether scripting elements are invalid in a group of JSP pages. By default, scripting is enabled.
`<page-encoding>`	Optional	Indicates pageEncoding information. It is a translation-time error to name different encodings in the pageEncoding attribute of the page directive of a JSP page and in a JSP configuration element matching the page. It is also a translation-time error to name different encodings in the prolog or text declaration of a document in XML syntax and in a JSP configuration element matching the document. It is legal to name the same encoding through multiple mechanisms.
`<is-xml>`	Optional	Indicates that a resource is a JSP document (XML). If true, denotes that the group of resources that match the URL pattern are JSP documents, and thus must be interpreted as XML documents. If false, the resources are assumed to not be JSP documents, unless there is another property group that indicates otherwise.
`<include-prelude>`	Optional	A context-relative path that must correspond to an element in the Web Application. When the element is present, the given path will be automatically included (as in an include directive) at the beginning of each JSP page in this jsp-property-group.
`<include-coda>`	Optional	A context-relative path that must correspond to an element in the Web application. When the element is present, the given path will be automatically included (as in an include directive) at the end of each JSP page in this `jsp-property-group`.
`<deferred-syntax-allowed-as-literal>`	Optional	Controls whether the character sequence `#{` is allowed when used as a String literal.

Element	Required/ Optional	Description
<trim-directive-whitespaces>	Optional	Controls whether template text containing only white spaces must be removed from the response output.
<url-pattern>	Required	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. Example patterns: /soda/grape/* /foo/* /contents *.foo The URL must follow the rules specified in the Servlet 2.4 Specification.

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-22.
<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 detailed instructions and an example on configuring security in Web applications, see *Securing WebLogic Resources*. Also, for more information on WebLogic Security, refer to *Programming WebLogic Security*.

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

Element	Required/ Optional	Description
<web-resource-collection>	Required	Defines the components of the Web application to which this security constraint is applied.
<auth-constraint>	Optional	Defines which groups or principals have access to the collection of web resources defined in this security constraint. See also "auth-constraint" on page A-20.
<user-data-constraint>	Optional	Defines how the client should communicate with the server. See also "user-data-constraint" on page A-21.

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-19.

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-19.

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-11.

user-data-constraint

This is an element within the "security-constraint" on page A-19.

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. (This is the default value.) `FORM`—uses a user-written HTML form. `CLIENT-CERT` **Note:** You can define multiple authentication methods as a comma separated list to provide a fall-back mechanism. Authentication will be attempted in the order the values are defined in the `auth-method` list. See Providing a Fallback Mechanism for Authentication Methods in *Programming WebLogic Security*.
`<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. **Note:** The `<realm-name>` element does not refer to system security realms within WebLogic Server. This element defines the realm name to use in HTTP Basic authorization. The system security realm is a collection of security information that is checked when certain operations are performed in the server. The servlet security realm is a different collection of security information that is checked when a page is accessed and basic authentication is used.
`<form-login-config>`	Optional	Use this element if you configure the `<auth-method>` to `FORM`. See "form-login-config" on page A-23.

form-login-config

This is an element within the "login-config" on page A-22.

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`-based authentication that conforms to a specific naming convention.
`<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-3.

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-6.
`<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.

Element	Required/ Optional	Description
<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 references the enterprise bean. The expected type of the referenced enterprise bean
- The expected local home and local interfaces of the referenced enterprise bean
- Optional ejb-link information, used to specify the referenced enterprise bean

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

Element	Required/ Optional	Description
<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>

Element	Required/ Optional	Description

`<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.

`<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 EJB.
		The name in the `ejb-link` element is composed of a path name. This path name specifies the `ejb-jar` containing the referenced EJB 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 EJB. This allows multiple EJBs with the same `ejb-name` to be uniquely identified.
		Used in: `ejb-local-ref` and `ejb-ref` elements.
		Examples:
		`<ejb-link>EmployeeRecord</ejb-link>`
		`<ejb-link>../products/product.jar#ProductEJB</ejb-link>`

web-app

The XML Schema for the Servlet 2.4 deployment descriptor. WebLogic Server fully supports HTTP servlets as defined in the Servlet 2.4 specification from Sun Microsystems. However, the `version` attributed must be set to `2.4` in order to enforce 2.4 behavior.

The following table describes the elements you can define within an `web-app` element.

Element	Required/ Optional	Description
`<version>`	Required	All Servlet deployment descriptors must indicate the 2.4 version of the schema in order to enforce Servlet 2.4 behavior.

APPENDIX B

weblogic.xml Deployment Descriptor Elements

This document provides a complete reference for the elements in the WebLogic Server-specific deployment descriptor `weblogic.xml`. If your Web application does not contain a `weblogic.xml` deployment descriptor, WebLogic Server automatically selects the default values of the deployment descriptor elements.

The following sections describe the complex deployment descriptor elements that can be defined in the `weblogic.xml` deployment descriptor under the root element `<weblogic-web-app>`:

weblogic.xml Namespace Declaration and Schema Location

The correct text for the namespace declaration and schema location for the WebLogic Server `weblogic.xml` file is as follows.

```
<weblogic-web-app xmlns="http://www.bea.com/ns/weblogic/weblogic-web-app">
```

To see the schema for `weblogic.xml`, go to http://www.bea.com/ns/weblogic/90/weblogic-web-app.xsd.

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 (as defined in the root element `<weblogic-web-app>`) 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 Web application security role and one or more principals in WebLogic Server, 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>
```

You can also use it to mark a given role as an externally defined role, as shown in the following example:

```
<security-role-assignment>
    <role-name>roleadmin</role-name>
    <externally-defined/>
</security-role-assignment>
```

Notes: In the `<security-role-assignment>` element, either `<principal-name>` or `<externally-defined>` must be defined. Both cannot be omitted.

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.

Element	Required Optional	Description
`<principal-name>`	Required if `<externally-defined>` is not defined.	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 *Managing WebLogic Security*.
`<externally-defined>`	Required if `<principal-name>` is not defined.	Specifies that a particular security role is defined globally in a security realm; WebLogic Server uses this security role as the principal name, rather than looking it up in a global realm. When the security role and its principal-name mapping are defined elsewhere, this is used as an indicative placeholder.

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

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

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

run-as-role-assignment

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

```
<run-as-role-assignment>
    <role-name>RunAsRoleName</role-name>
```

```
    <run-as-principal-name>joe</run-as-principal-name>
</run-as-role-assignment>
```

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

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

resource-description

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

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

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

resource-env-description

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

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

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

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.

service-reference-description

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

Element	Required/ Optional	Description
`<service-ref-name>`		
`<wsdl-url>`		

Element	Required/ Optional	Description
<call-property>		The <call-property> element has the following sub-elements: • <name> • <value>
<port-info>		The <port-info> element has the following sub-elements: • <port-name> • <stub-property> • <call-property>

session-descriptor

The session-descriptor elements that define parameters for servlet sessions.

Note: When initializing session context, most session descriptors from weblogic-application.xml take precedence over those from weblogic.xml, with the default value being used for undefined properties regardless if it exists in weblogic.xml. However, when both XML files are being used, the following properties in weblogic.xml are honored first:

- debug-enabled
- cache-size
- cookie-max-age-secs
- timeout-secs
- max-in-memory-sessions
- monitoring-attribute-name
- invalidation-interval-secs

Element Name	Default Value	Value
`timeout-secs`	3600	Sets the time, in seconds, that WebLogic Server waits before timing out a session. The default value is 3600 seconds. 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 element can be overridden by the `session-timeout` element (defined in minutes) in `web.xml`.
`invalidation-interval-secs`	`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 element to tune WebLogic Server for best performance on high traffic sites. The default value is 60 seconds.
`sharing-enabled`	`false`	Enables Web applications to share HTTP sessions when the value is set to `true` at the application level. This element is ignored if turned on at the Web application level.
`debug-enabled`	`false`	Enables the debugging feature for HTTP sessions. The default value is `false`.

Element Name	Default Value	Value
`id-length`	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 attributes), which limits the amount of data that can be transmitted using URL re-writing. To allow more space for attributes, use this attribute to limit the size of the session ID that is randomly generated by WebLogic Server. You can also limit the length to a fixed 52 characters, and disallow special characters, by setting the WAPEnabled attribute. For more information, see URL Rewriting and Wireless Access Protocol in Developing Web Applications for WebLogic Server.
`tracking-enabled`	`true`	Enables session tracking between HTTP requests.
`cache-size`	`1028`	Sets the cache size for JDBC and file-persistent sessions.

Element Name	Default Value	Value
`max-in-memory-sessions`	`-1`	Sets the maximum limit for memory/replicated sessions. Without the ability to configure bound in-memory servlet session use, as new sessions are continually created, the server eventually throws out of memory. To protect against this, WebLogic Server provides a configurable bound on the number of sessions created. When this number is exceeded, the `weblogic.servlet.SessionCreationException` occurs for each attempt to create a new session. This feature applies to both replicated and non-replicated in-memory sessions. To configure bound in-memory servlet session use, you set the limitation in the `max-in-memory-sessions` element. The default is `-1` (unlimited).
`cookies-enabled`	`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.
`cookie-name`	`JSESSIONID`	Defines the session tracking cookie name. Defaults to `JSESSIONID` if not set. You may set this to a more specific name for your application.
`cookie-path`	`null`	Defines the session tracking cookie path. If not set, this attribute 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.

Element Name	Default Value	Value
`cookie-domain`	`null`	Specifies the domain for which the cookie is valid. For example, setting `cookie-domain` 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 not set, this attribute defaults to the server that issued the cookie. For more information, see `Cookie.setDomain()` in the Servlet specification from Sun Microsystems.
`cookie-comment`	`null`	Specifies the comment that identifies the session tracking cookie in the cookie file.
`cookie-secure`	`false`	Tells the browser to only send the cookie back over an HTTPS connection. This ensures that the cookie ID is secure and should only be used on Websites that use HTTPS. Session Cookies over HTTP no longer work if this feature is enabled. You should disable the `url-rewriting-enabled` element if you intend to use this feature.
`cookie-max-age-secs`	`-1`	Sets the life span of the session cookie, in seconds, after which it expires on the client. The default value is -1 (unlimited) For more information about cookies, see "Using Sessions and Session Persistence" on page 8-1.

Element Name	Default Value	Value
`persistent-store-type`	`memory`	Sets the persistent store method to one of the following options: • `memory`—Disables persistent session storage. • `replicated`—Same as `memory`, but session data is replicated across the clustered servers. • `replicated_if_clustered`—If the Web application is deployed on a clustered server, the in-effect `persistent-store-type` will be replicated. Otherwise, `memory` is the default. • `sync-replication-across-cluster`—The replication will occur synchronously across the cluster. • `async-replication-across-cluster`—The replication will occur asynchronously across the cluster. • `file`—Uses file-based persistence (See also "`persistent-store-dir`" on page B-13). • `jdbc`—Uses a database to store persistent sessions. (see also "`persistent-store-pool`" on page B-13). • `cookie`—All session data is stored in a cookie in the user's browser.
`persistent-store-cookie-name`	`WLCOOKIE`	Sets the name of the cookie used for cookie-based persistence. The `WLCOOKIE` cookie carries the session state, which should not be shared between Web applications. For more information, see "Using Cookie-Based Session Persistence" on page 8-11.

Element Name	Default Value	Value
`persistent-store-dir`	`session_db`	Specifies the storage directory used for file-based persistence 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 `persistent-store-dir`. Note that the size of each session can vary as the size of serialized session data changes. Each server instance has a default persistent file store that requires no configuration. Therefore, if no directory is specified, a default store is automatically created in the `<server-name>\data\store\default` directory. However, the default store is not shareable among clustered servers. You can make file-persistent sessions clusterable by creating a custom persistent store in a directory that is shared among different servers. However, this requires you to create this directory manually.
`persistent-store-pool`	`None`	Specifies the name of a JDBC connection pool to be used for persistence storage.
`persistent-store-table`	`wl_servlet_sessions`	Specifies the database table name used to store JDBC-based persistent sessions. This applies only when `persistent-store-type` is set to `jdbc`. The `persistent-store-table` element is used when you choose a database table name other than the default.
`jdbc-column-name-max-inactive-interval`		Serves as an alternative name for the `wl_max_inactive_interval` column name. This `jdbc-column-name-max-inactive-interval` element applies only to JDBC-based persistence. It is required for certain databases that do not support long column names.

Element Name	Default Value	Value
`jdbc-connection-timeout-secs`	120	Note: This is a deprecated item for this release. Sets the time, in seconds, that WebLogic Server waits before timing out a JDBC connection, where x is the number of seconds between.
`url-rewriting-enabled`	`true`	Enables URL rewriting, which encodes the session ID into the URL and provides session tracking if cookies are disabled in the browser.
`http-proxy-caching-of-cookies`	`true`	When set to `false`, WebLogic Server adds the following header with the following response: `"Cache-control: no-cache=set-cookie"` This indicates that the proxy caches do not cache the cookies.
`encode-session-id-in-query-params`	`false`	The latest servlet specification requires containers to encode the session ID in path parameters. Certain Web servers do not work well with path parameters. In such cases, the `encode-session-id-in-query-params` element should be set to `true`. (The default is `false`.)
`runtime-main-attribute`		Used in `ServletSessionRuntimeMBean`. The `getMainAttribute()` of the `ServletSessionRuntimeMBean` returns the session attribute value using this string as a key. Example: `user-name` This element is useful for tagging session runtime information for different sessions.
`cookie-http-only`	`true`	Specifies whether HttpOnly cookies are enabled. When this element is set to `true`, all session cookies would be unavailable to the browser scripts. The default value is `true`. Therefore, HttpOnly cookies are enabled by default.

jsp-descriptor

The `jsp-descriptor` element specifies a list of configuration parameters for the JSP compiler. The following table describes the elements you can define within a `jsp-descriptor` element.

Element	Required/ Optional	Description
`page-check-seconds`	`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. • The value `-1` means never check the pages. This is the default value in a production environment. • The value `0` means always check the pages. • The value `1` means check the pages every second. This is the default value in a development environment. In a production environment where changes to a JSP are rare, consider changing the value of pageCheckSeconds to 60 or greater, according to your tuning requirements.
`precompile`	`false`	When set to true, WebLogic Server automatically precompiles all modified JSPs when the Web application is deployed or re-deployed or when starting WebLogic Server.
`precompile-continue`	`false`	When set to true, WebLogic Server continues precompiling all modified JSPs even if some of those JSPs fail during compilation. Only takes effect when precompile is set to true.
`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.
`verbose`	`true`	When set to `true`, debugging information is printed out to the browser, the command prompt, and WebLogic Server log file.
`working-dir`	internally generated directory	The name of a directory where WebLogic Server saves the generated Java and compiled class files for a JSP. Note: If `weblogic.xml` defines a `working-dir`, WebLogic Server does not delete this directory when the web application is undeployed.

Element	Required/ Optional	Description
`print-nulls`	`null`	When set to false, this parameter ensures that expressions with "null" results are printed as " ".
`backward-compatible`	`true`	When set to true, backward compatibility is enabled.
`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 attribute 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"%>`
`package-prefix`	`jsp_servlet`	Specifies the package prefix into which all JSP pages are compiled.
`exact-mapping`	`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 Web application container generates non-exact url mapping for JSPs. `exactMapping` allows path info for JSP pages.
`default-file-name`	`true`	The default file name in which WebLogic Server saves the generated Java and compiled class files for a JSP.
`rtexprvalue-jsp-param-name`	`false`	Allows runtime expression values in the name attribute of the `jsp:param` tag. It is set to `false` by default.

auth-filter

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

Note: This is a deprecated element for the current release. Instead, use servlet authentication filters.

container-descriptor

The <container-descriptor> element specifies a list of parameters that affect the behavior of the Web application.

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: As a best practice, BEA does not recommend that you enable the `check-auth-on-forward` property.

filter-dispatched-requests-enabled

The <filter-dispatched-requests-enabled> element controls whether or not filters are applied to dispatched requests. The default value is `false`.

Note: Because 2.4 servlets are backward compatible with 2.3 servlets (per the 2.4 specification), when 2.3 descriptor elements are detected by WebLogic Server, the <filter-dispatched-requests-enabled> element defaults to `true`.

redirect-with-absolute-url

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

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

user readable data used in a redirect.

index-directory-enabled

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

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

index-directory-sort-by

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

servlet-reload-check-secs

The `<servlet-reload-check-secs>` element defines whether a WebLogic Server will check to see if a servlet has been modified, and if it has been modified, reloads it.

- The value `-1` means never check the servlets. This is the default value in a production environment.
- The value `0` means always check the servlets.
- The value `1` means check the servlets every second. This is the default value in a development environment.

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

resource-reload-check-secs

The `<resource-reload-check-secs>` element is used to perform metadata caching for cached resources that are found in the resource path in the Web application scope. This parameter identifies how often WebLogic Server checks whether a resource has been modified and if so, it reloads it.

- The value `-1` means metadata is cached but never checked against the disk for changes. In a production environment, this value is recommended for better performance.
- The value `0` indicates not to do any metadata caching. Customers who keep changing their files must set this parameter to a value greater than or equal to 0.
- The value `1` means reload every second. This is the default value in a development environment.

Values specified for this parameter using the Admin Console are given precedence.

single-threaded-servlet-pool-size

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

Note: SingleThreadMode instance pools are deprecated in this release.

session-monitoring-enabled

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

save-sessions-enabled

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

prefer-web-inf-classes

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

default-mime-type

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

client-cert-proxy-enabled

The <client-cert-proxy-enabled> element default value is `true`. When set to `true`, WebLogic Server passes identity certificates from the clients to the backend servers. Also, WebLogic Server is notified whether to honor or discard the incoming WL-Proxy-Client-Cert header.

A proxy-server plugin encodes each identity certification in the WL-Proxy-Client-Cert header and passes it to the backend WebLogic Server instances. Each WebLogic Server instance takes

the certificate information from the header, ensured it came from a secure source, and uses that information to authenticate the user. For the background WebLogic Server instances, this parameter must be set to `true` (either at the cluster/server level or at the Web application level).

If you set this element to `true`, use a weblogic.security.net.ConnectionFilter to ensure that each WebLogic Server instance accepts connections only from the machine on which the proxy-server plugin is running. If you specify `true` without using a connection filter, a potential security vulnerability is created because the WL-Proxy-Client-Cert header can be spoofed.

relogin-enabled

The `<relogin-enabled>` element is a backward compatibility parameter. If a user has logged in already and tries to access a resource for which s/he does not have privileges, a `FORBIDDEN (403)` response occurs.

allow-all-roles

In the security-constraints elements defined in `web.xml` descriptor of a Web application, the `auth-constraint` element indicates the user roles that should be permitted access to this resource collection. Here role-name = "*" is a compact syntax for indicating all roles in the Web application. In past releases, role-name = "*" was treated as all users/roles defined within the realm.

This `allow-all-roles` element is a backward compatibility switch to restore old behavior. The default behavior is to allow all roles defined in the Web application. The value specified in `weblogic-xml` takes precedence over the value defined in the `WebAppContainerMBean`.

native-io-enabled

To use native I/O while serving static files with `weblogic.servlet.FileServlet`, which is implicitly registered as the default servlet, set `native-io-enabled` to `true`. (The default value is `false`.) `native-io-enabled` element applies only on Windows.

minimum-native-file-size

The `minimum-native-file-size` element applies only when `native-io-enabled` is set to `true`. It sets the minimum file size for using native I/O. If the file being served is larger than this value, native I/O is used. If you do not set this value, the default value used is `4K`.

disable-implicit-servlet-mapping

When the `disable-implicit-servlet-mappings` flag is set to `true`, the Web application container does not create implicit mappings for internal servlets (`*.jsp`, `*.class`, and so on); only for the default servlet mapping. A typical use case for turning off implicit servlet mappings would be when configuring `HttpClusterServlet` or `HttpProxyServlet`.

The default value is `false`.

optimistic-serialization

When `optimistic-serialization` is turned on, WebLogic Server does not serialize-deserialize context and request attributes upon `getAttribute(name)` when the request is dispatched across servlet contexts.

This means that you must make sure that the attributes common to Web applications are scoped to a common parent classloader (application scoped) or you must place them in the system classpath if the two Web applications do not belong to the same application.

When `optimistic-serialization` is turned off (default value), WebLogic Server serialize-deserializes context and request attributes upon `getAttribute(name)` to avoid the possibility of ClassCastExceptions.

The `optimistic-serialization` value can also be specified at domain level in the `WebAppContainerMBean`, which applies for all Web applications. The value in `weblogic.xml`, if specified, overrides the domain level value.

The default value is `false`.

The `WebAppComponentRuntimeBean.getSessionIds()` method returns an array of session attribute values with this name. If it is not set, it returns an array of randomly generated Strings.

require-admin-traffic

The `require-admin-trafffic` element defines whether traffic should go through the administration channel. When set to `true` traffic is allowed to go through the administration channel. Otherwise, traffic can only go through administration channel when the Web application is in administrative mode. For example:

```
<container-descriptor>
    <require-admin-traffic>true</require-admin-traffic>
</container-descriptor>
```

charset-params

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

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

input-charset

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

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

For more information, see "Determining the Encoding of an HTTP Request" on page 5-5.

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 5-6.

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.

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

url-match-map

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

Rule for `SimpleApacheURLMatchMap`:

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

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

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

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

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 `META-INF/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.

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

1. Check `application.xml` for context root; if found, use as Web application's context root.
2. If context root is not set in `application.xml`, and the Web application is being deployed as part of an EAR, check whether context root is defined in `weblogic.xml`. If found, use as Web application's context root. If the Web application is deployed standalone, `application.xml` does not come into play and the determination for context-root starts at `weblogic.xml` and defaults to URI if it is not defined there.
3. If context root is not defined in `weblogic.xml` or `application.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`.

Note: The `context-root` element cannot be set for individual Web applications in EAR libraries. It can only bet set for Web application libraries.

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 parameter can be overridden at the individual servlet or jsp level by using the `per-servlet-dispatch-policy` element.

servlet-descriptor

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

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

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

work-manager

The `work-manager` element is a sub-element of the `<weblogic-web-app>` element. You can define the following elements within the `work-manager` element.

Element	Required Optional	Description
`name`	Required	Specifies the name of the Work Manager.
`response-time-request-class / fair-share-request-class / context-request-class / request-class-name`	Optional	You can choose between the following four elements: `response-time-request-class`—Defines the response time request class for the application. Response time is defined with attribute goal-ms in milliseconds. The increment is ((goal –T) C_r)/R, where T is the average thread use time, R the arrival rate, and C_r a coefficient to prioritize response time goals over fair shares. `fair-share-request-class`—Defines the fair share request class. Fair share is defined with attribute percentage of default share. Therefore, the default is 100. The increment is C_f/(P R T), where P is the percentage, R the arrival rate, T the average thread use time, and C_f a coefficient for fair shares to prioritize them lower than response time goals. `context-request-class`—Defines the context class. Context is defined with multiple cases mapping contextual information, like current user or its role, cookie, or work area fields to named service classes. `request-class-name`—Defines the request class name.

Element	Required Optional	Description
`min-threads-constraint`, `min-threads-constraint-name`	Optional	You can choose between the following two elements: `min-threads-constraint`—Used to guarantee a number of threads the server allocates to requests of the constrained work set to avoid deadlocks. The default is zero. A min-threads value of one is useful, for example, for a replication update request, which is called synchronously from a peer. `min-threads-constraint-name`—Defines a name for the `min-threads-constraint` element.

Element	Required Optional	Description
`max-threads-constraint`, `max-threads-constraint-name`	Optional	You can choose between the following two elements: `max-threads-constraint`—Limits the number of concurrent threads executing requests from the constrained work set. The default is unlimited. For example, consider a constraint defined with maximum threads of 10 and shared by 3 entry points. The scheduling logic ensures that not more than 10 threads are executing requests from the three entry points combined. `max-threads-constraint-name`—Defines a name for the `max-threads-constraint` element.
`capacity, capacity-name`	Optional	You can choose between the following two elements: `capacity`—Constraints can be defined and applied to sets of entry points, called constrained work sets. The server starts rejecting requests only when the capacity is reached. The default is zero. Note that the capacity includes all requests, queued or executing, from the constrained work set. This constraint is primarily intended for subsystems like JMS, which do their own flow control. This constraint is independent of the global queue threshold. `capacity-name`—Defines a name for the `capacity` element.

logging

The `logging` element is a sub-element of the `<weblogic-web-app>` element. You can define the following elements within the `logging` element.

Element	Required Optional	Description
`log-filename`	Required	Specifies the name of the log file. The full address of the filename is required.
`logging-enabled`	Optional	Indicates whether or not the log writer is set for either the `ManagedConnectionFactory` or `ManagedConnection`. If this element is set to true, output generated from either the `ManagedConnectionFactory` or `ManagedConnection` will be sent to the file specified by the `log-filename` element. Failure to specify this value will result in WebLogic Server using its defined default value. Value Range: `true` | `false` Default Value: `false`

Element	Required Optional	Description
`rotation-type`	Optional	Sets the file rotation type. Values are `bySize, byName, none` • `bySize`—When the log file reaches the size that you specify in `file-size-limit`, the server renames the file as *`FileName.n`*. • `byName`—At each time interval that you specify in `file-time-span`, the server renames the file as *`FileName.n`*. After the server renames a file, subsequent messages accumulate in a new file with the name that you specified in `log-filename`. • `none`—Messages accumulate in a single file. You must erase the contents of the file when the size is unwieldy. Default Value: `bySize`

Element	Required Optional	Description
`number-of-files-limited`	Optional	Specifies whether the number of files that this server instance creates to store old messages should be limited. (Requires that you specify a `rotation-type` of `bySize`). After the server reaches this limit, it overwrites the oldest file. If you do not enable this option, the server creates new files indefinitely and you must clean up these files as you require. If you enable `number-of-files-limited` by setting it to `true`, the server refers to your `rotationType` variable to determine how to rotate the log file. *Rotate* means that you override your existing file instead of creating a new file. If you specify `false` for `number-of-files-limited`, the server creates numerous log files rather than overriding the same one. Value Range: `true` | `false` Default Value: `false`
`file-count`	Optional	The maximum number of log files that the server creates when it rotates the log. This number does not include the file that the server uses to store current messages. (Requires that you enable `number-of-files-limited`.) Default Value: `7`
`file-size-limit`	Optional	The size that triggers the server to move log messages to a separate file. (Requires that you specify a `rotation-type` of `bySize`.) After the log file reaches the specified minimum size, the next time the server checks the file size, it will rename the current log file as *`FileName.n`* and create a new one to store subsequent messages. Default Value: `500`

Element	Required Optional	Description
`rotate-log-on-startup`	Optional	Specifies whether a server rotates its log file during its startup cycle. Value Range: `true` | `false` Default Value: `true`
`log-file-rotation-dir`	Optional	Specifies the directory path where the rotated log files will be stored.
`rotation-time`	Optional	The start time for a time-based rotation sequence of the log file, in the format `k:mm`, where `k` is 1-24. (Requires that you specify a `rotation-type` of `byTime`.) At the specified time, the server renames the current log file. Thereafter, the server renames the log file at an interval that you specify in `file-time-span`. If the specified time has already past, then the server starts its file rotation immediately. By default, the rotation cycle begins immediately.
`file-time-span`	Optional	The interval (in hours) at which the server saves old log messages to another file. (Requires that you specify a `rotation-type` of `byTime`.) Default Value: `24`

library-ref

The `library-ref` element references a library module, which is intended to be used as a Web application library in the current Web application.

Example:

```
<library-ref>
     <library-name>WebAppLibraryFoo</library-name>
     <specification-version>2.0</specification-version>
     <implementation-version>8.1beta</implementation-version>
```

```
    <exact-match>false</exact-match>
</library-ref>
```

Only the following sub-elements are relevant to Web applications: `library-name`, `specification-version`, `implementation-version`, and `exact-match`.

You can define the following elements within the `library-ref` element.

Element	Required Optional	Description
`library-name`	Required	Provides the library name for the library module reference. The default value is `null`.
`specification-version`	Required	Provides the specification version for the library module reference. The default value is `0`. (This is a float.)
`implementation-version`	Required	Provides the implementation version for the library module reference. The default value is `null`.
`exact-match`	Required	The default value is `false`.

Backwards Compatibility Flags

Several backwards compatibility flags have been added to allow you to restore behavior seen in releases prior to WebLogic Server 9.0. For a complete list and description of these flags and for all information about Web Application, JSP, and Servlet backwards compatibility, see Compatibility with Previous Releases in *Upgrading WebLogic Application Environments*.

Web Container Global Configuration

To configure your Web container at a global level, use the WebAppContainerMBean. For information on the WebAppContainerMBean attributes and how to use them to specify domain-wide defaults for all of your Web applications, see the WebAppContainerMBean at http://e-docs.bea.com/wls/docs90/wlsmbeanref/mbeans/WebAppContainerMBean.html.

APPENDIX C

Web Application Best Practices

The following sections contain BEA best practices for designing, developing, and deploying WebLogic Web applications and application resources:

CGI Best Practices

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

- You can use `sh subscript.sh` for both exploded (unarchived) Web applications and archived Web applications (WAR files).
- You can use `sh $PWD/subscript.sh` for both exploded (unarchived) Web applications and archived Web applications (WAR files).
- You can use `sh $DOCUMENT_ROOT/$PATH/subscript.sh` for exploded (unarchived) Web applications. You cannot use it, however, for archived Web applications (WAR files). This is due to the fact that the document root might point you to the root of your WAR file, and the scripting language cannot open that WAR file and locate the `subscript.sh` needed for execution. This is true not only for `sh`, but for any scripting language.

Servlet Best Practices

Consider the following best practices when writing HTTP servlets:

- Compile your servlet classes into the `WEB-INF/classes` directory of your Web Application.
- Make sure your servlet is registered in the J2EE standard Web applications deployment descriptor (`web.xml`).
- When responding to a request for a servlet, WebLogic Server checks the time stamp of the servlet class file prior to applying any filters associated with the servlet, and compares it to the servlet instance in memory. If a newer version of the servlet class is found, WebLogic Server re-loads all servlet classes before any filtering takes place. When the servlets are re-loaded, the `init()` method of the servlet is called. All servlets are reloaded when a modified servlet class is discovered due to the possibility that there are interdependencies among the servlet classes.

  You can set the interval (in seconds) at which WebLogic Server checks the time stamp with the `Servlet Reload` attribute. This attribute is set on the `Files` tab of your Web Application, in the Administration Console. If you set this attribute to zero, WebLogic Server checks the time stamp on every request, which can be useful while developing and testing servlets but is needlessly time consuming in a production environment. If this attribute is set to `-1`, WebLogic Server does not check for modified servlets.

JSP Best Practices

For a complete explanation on how to avoid JSP recompilation, see Avoiding Unnecessary JSP Compilation and specifically the section called Scenarios that Cause Recompilation of JSPs.

Best Practice When Subclassing ServletResponseWrapper

J2EE provides the class `javax.servlet.ServletResponseWrapper`, which you can subclass in your Servlet to adapt its response.

BEA recommends that if you create your own response wrapper by subclassing the `ServletResponseWrapper` class, you should always override the `flushBuffer()` and `clearBuffer()` methods. Not doing so might result in the response being committed prematurely.