Oracle® Fusion Middleware

Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server



26 (26.1.0.0.0)
G56909-01
August 2026



Oracle Fusion Middleware Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server, 26 (26.1.0.0.0)

G56909-01

Copyright © 2007, 2026, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation ii
Conventions ii

1 Understanding Web Applications, Servlets, and JSPs

The Web Applications Container 1
- Web Applications and Jakarta EE 1
- Web Application Development Key Points 1

Servlets 2
- Servlets and Jakarta EE 2
- What You Can Do with Servlets 3
- Servlet Development Key Points 3

Jakarta Server Pages 3
- JSPs and Jakarta EE 4
- What You Can Do with JSPs 4
- Overview of How JSP Requests Are Handled 4

Web Application Developer Tools 5
- Other Tools 5

Web Application Security 5
- Limiting the Number of Parameters in an HTTP Request 6

Avoiding Redirection Attacks 6
Using HTTP Strict Transport Security 7

2 Creating and Configuring Web Applications

WebLogic Web Applications and Jakarta EE 1
Directory Structure 1
- Accessing Information in WEB-INF 2
- Directory Structure Example 2

Main Steps to Create and Configure a Web Application 2

Copyright © 2007, 2026, Oracle and/or its affiliates.

Step One: Create the Enterprise Application Wrapper 2
Step Two: Create the Web Application 3
Step Three: Creating the build.xml File 3
Step Four: Execute the Split Development Directory Structure Ant Tasks 4
Configuring How a Client Accesses a Web Application 4
Configuring Virtual Hosts for Web Applications 4
Configuring a Channel-based Virtual Host 5
Configuring a Host-based Virtual Host 5
Targeting Web Applications to Virtual Hosts 5
Loading Servlets, Context Listeners, and Filters 5
Shared Jakarta EE Web Application Libraries 6
Enabling GZIP Compression for Web Applications 6

3 Using Jakarta MVC 3.0 in WebLogic Server Web Applications

Developing Jakarta MVC Applications 1
Understanding Jakarta MVC Applications 2
Using Models, Views, and Controllers 5
Using Data Binding in Jakarta MVC Applications 8
Securing Jakarta MVC Applications 10
Using Events in Jakarta MVC Applications 12
Configuring Jakarta MVC Applications 14
Additional Jakarta MVC Capabilities 16
Using View Engines in Jakarta MVC Applications 16
Internationalizing Jakarta MVC Applications 19
Using Form Method Override in Jakarta MVC Applications 20
Jakarta MVC Implementation Reference for Oracle WebLogic Server 22

4 Creating and Configuring Servlets

What's New and Changed in Servlets in 26.1.0.0.0 1
What Was New and Changed in Servlet 6.1 1
Configuring Servlets 1
Servlet Annotations 1
Servlet Mapping 2
Setting Up a Default Servlet 3
Servlet Initialization Attributes 4
Writing a Simple HTTP Servlet 4
Advanced Features 6
Complete HelloWorldServlet Example 6
Debugging Servlet Containers 7
Disabling Access Logging 7

Copyright © 2007, 2026, Oracle and/or its affiliates.

Usage 7
Example 8
Debugging Specific Sessions 8
Usage 8
Tracking a Request Handle Footprint 8
Usage 9

5 Creating and Configuring JSPs

Configuring Jakarta Server Pages (JSPs) 1
Registering a JSP as a Servlet 1
Configuring JSP Tag Libraries 2
Configuring Welcome Files 2
Customizing HTTP Error Responses 3
Determining the Encoding of an HTTP Request 3
Mapping IANA Character Sets to Java Character Sets 4
Configuring Implicit Includes at the Beginning and End of JSPs 4
Configuring JSP Property Groups 5
JSP Property Group Rules 5
What You Can Do with JSP Property Groups 5
Writing JSP Documents Using XML Syntax 6
How to Use JSP Documents 6
Important Information about JSP Documents 6

6 Using JSF and JSTL

Using JSF and JSTL With Web Applications 1
Jakarta Server Faces (JSF) 1
Jakarta Standard Tag Library (JSTL) 2
JSF Backward Compatibility 2
Deploying JSF and JSTL Libraries 3
Referencing a JSF or JSTL Library 4

7 Configuring Resources in a Web Application

Configuring Resources in a Web Application 1
Configuring Resources 1
Referencing External EJBs 2
More about the ejb-ref* Elements 3
Referencing Application-Scoped EJBs 3
Serving Resources from the CLASSPATH with the ClasspathServlet 5
Using CGI with WebLogic Server 6

Copyright © 2007, 2026, Oracle and/or its affiliates.

Configuring WebLogic Server to Use CGI 7
Requesting a CGI Script 7
CGI Best Practices 8

8 WebLogic Annotation for Web Components

Servlet Annotation and Dependency Injection 1
Web Component Classes That Support Annotations 2
Annotations Supported By a Web Container 3
Fault Detection and Recovery 4
Limitations 4
Deprecated WebLogic Server-Specific Servlet Annotations 4
WLServlet 4
Attributes 5
Fault Detection And Recovery 6
WLFilter 6
Attributes 6
Fault Detection and Recovery 7
WLInitParam 7
Attributes 7

9 Servlet Programming Tasks

Initializing a Servlet 1
Initializing a Servlet when WebLogic Server Starts 1
Overriding the init() Method 2
Providing an HTTP Response 2
Retrieving Client Input 4
Methods for Using the HTTP Request 5
Example: Retrieving Input by Using Query Parameters 5
Securing Client Input in Servlets 6
Compatibility: Using a WebLogic Server Utility Method 7
Using Cookies in a Servlet 7
Setting Cookies in an HTTP Servlet 7
Retrieving Cookies in an HTTP Servlet 7
Using Cookies That Are Transmitted by Both HTTP and HTTPS 8
Application Security and Cookies 8
Response Caching 9
Initialization Parameters 9
Using WebLogic Services from an HTTP Servlet 10
Accessing Databases 10
Connecting to a Database Using a DataSource Object 10

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using a Data Source in a Servlet 11
Connecting Directly to a Database Using a JDBC Driver 11
Threading Issues in HTTP Servlets 11
Dispatching Requests to Another Resource 12
Forwarding a Request 12
Including a Request 13
RequestDispatcher and Filters 13
Proxying Requests to Another Web Server 13
Overview of Proxying Requests to Another Web Server 14
Setting Up a Proxy to a Secondary Web Server 14
Sample Deployment Descriptor for the Proxy Servlet 15
Proxy Servlet Parameters 16
Clustering Servlets 16
Referencing a Servlet in a Web Application 17
URL Pattern Matching 17
The SimpleApacheURLMatchMap Utility 18
A Future Response Model for HTTP Servlets 18
Abstract Asynchronous Servlet 18
doRequest 18
doResponse 19
doTimeOut 19
Future Response Servlet 20

10 Using Sessions and Session Persistence

Overview of HTTP Sessions 1
Setting Up Session Management 1
HTTP Session Properties 1
Session Timeout 1
Configuring WebLogic Server Session Cookies 2
Configuring Application Cookies That Outlive a Session 2
Logging Out 2
Enabling Web Applications to Share the Same Session 3
Limiting Number of Concurrent Requests for a Session 3
Configuring Session Persistence 3
Attributes Shared by Different Types of Session Persistence 4
Using Memory-based, Single-server, Non-replicated Persistent Storage 4
Using File-based Persistent Storage 4
Using a Database for Persistent Storage (JDBC Persistence) 5
Configuring JDBC-based Persistent Storage 5
Caching and Database Updates for JDBC Session Persistence 7
Using Cookie-Based Session Persistence 7

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using URL Rewriting Instead of Cookies 8
Coding Guidelines for URL Rewriting 8
URL Rewriting and Wireless Access Protocol (WAP) 9
Session Tracking from a Servlet 10
A History of Session Tracking 10
Tracking a Session with an HttpSession Object 10
Lifetime of a Session 11
How Session Tracking Works 12
Detecting the Start of a Session 12
Setting and Getting Session Name/Value Attributes 12
Logging Out and Ending a Session 13
Using session.invalidate() for a Single Web Application 13
Implementing Single Sign-On for Multiple Applications 13
Exempting a Web Application for Single Sign-on 14
Configuring Session Tracking 14
Using URL Rewriting Instead of Cookies 14
URL Rewriting and Wireless Access Protocol (WAP) 15
Making Sessions Persistent 15
Scenarios to Avoid When Using Sessions 15
Use Serializable Attribute Values 16
Configuring Session Persistence 16
Configuring a Maximum Limit on In-memory Servlet Sessions 16
Enabling Session Memory Overload Protection 16

11 Application Events and Event Listener Classes

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

12 WebLogic JSP Reference

JSP Tags 1
Defining JSP Versions 2
Rules for Defining a JSP File Version 2

Copyright © 2007, 2026, Oracle and/or its affiliates.

Rules for Defining a Tag File Version 3
Reserved Words for Implicit Objects 3
Directives for WebLogic JSP 4
Using the page Directive to Set Character Encoding 5
Using the taglib Directive 5
Declarations 5
Scriptlets 6
Expressions 6
Compatibility Example: JSP with Embedded Java 7
Actions 8
Using JavaBeans in JSP 8
Instantiating the JavaBean Object 8
Doing Setup Work at JavaBean Instantiation 8
Using the JavaBean Object 9
Defining the Scope of a JavaBean Object 9
Forwarding Requests 9
Including Requests 10
JSP Expression Language 10
Expressions and Attribute Values 10
Expressions and Template Text 11
JSP Expression Language Implicit Objects 12
JSP Expression Language Literals and Operators 13
Literals 13
Errors, Warnings, Default Values 13
Operators 13
Operator Precedence 14
JSP Expression Language Reserved Words 14
JSP Expression Language Named Variables 15
Securing User-Supplied Data in JSPs 15
Compatibility: Using a WebLogic Server Utility Method 16
Using Sessions with JSP 16
Deploying Applets from JSP 17
Using the WebLogic JSP Compiler 18
JSP Compiler Syntax 18
JSP Compiler Options 18
Precompiling JSPs 20
Using the JSPClassServlet 20

13 Filters

Overview of Filters 1
How Filters Work 1

Copyright © 2007, 2026, Oracle and/or its affiliates.

Uses for Filters 1
Writing a Filter Class 2
Configuring Filters 2
Configuring a Filter 2
Configuring a Chain of Filters 4
Filtering the Servlet Response Object 4
Additional Resources 4

14 Using WebLogic JSP Form Validation Tags

Overview of WebLogic JSP Form Validation Tags 1
Validation Tag Attribute Reference 1
<wl:summary> 1
<wl:form> 2
<wl:validator> 3
Using WebLogic JSP Form Validation Tags in a JSP 3
Creating HTML Forms Using the <wl:form> Tag 4
Defining a Single Form 4
Defining Multiple Forms 5
Re-Displaying the Values in a Field When Validation Returns Errors 5
Re-Displaying a Value Using the <input> Tag 5
Re-Displaying a Value Using the Apache Jakarta <input:text> Tag 5
Using a Custom Validator Class 6
Extending the CustomizableAdapter Class 6
Sample User-Written Validator Class 7
Sample JSP with Validator Tags 7

15 Using Custom WebLogic JSP Tags (cache, process, repeat)

Overview of WebLogic Custom JSP Tags 1
Using the WebLogic Custom Tags in a Web Application 1
Cache Tag 2
Refreshing a Cache 2
Flushing a Cache 2
Process Tag 6
Repeat Tag 7

16 Using the WebLogic EJB to JSP Integration Tool

Overview of the WebLogic EJB-to-JSP Integration Tool 1
Basic Operation 2
Interface Source Files 2

Copyright © 2007, 2026, Oracle and/or its affiliates.

Build Options Panel 3
Troubleshooting 3
Using EJB Tags on a JSP Page 4
EJB Home Methods 4
Stateful Session and Entity Beans 4
Default Attributes 5

A web.xml Deployment Descriptor Elements

web.xml Namespace Declaration and Schema Location A-1
context-param A-1
description A-3
display-name A-3
distributable A-3
ejb-local-ref A-4
ejb-ref A-5
env-entry A-5
error-page A-6
filter A-7
filter-mapping A-7
icon A-8
jsp-config A-9
- taglib A-9
- jsp-property-group A-10

listener A-11
login-config A-12
- form-login-config A-12

message-destination-ref A-13
mime-mapping A-14
resource-env-ref A-14
resource-ref A-15
security-constraint A-16
- web-resource-collection A-16
- auth-constraint A-17
- user-data-constraint A-17

security-role A-18
servlet A-18
- icon A-19
- init-param A-20
- security-role-ref A-20

servlet-mapping A-21
session-config A-22

Copyright © 2007, 2026, Oracle and/or its affiliates.

web-app A-23
welcome-file-list A-23

B weblogic.xml Deployment Descriptor Elements

weblogic.xml Namespace Declaration and Schema Location B-1
async-descriptor B-1
async-work-manager B-1
auth-filter B-2
charset-params B-2
- charset-mapping B-2
- input-charset B-3

container-descriptor B-3
- access-logging-disabled B-3
- allow-all-roles B-3
- check-auth-on-forward B-3
- client-cert-proxy-enabled B-4
- container-initializer-enabled B-4
- default-mime-type B-5
- disable-implicit-servlet-mappings B-5
- filter-dispatched-requests-enabled B-5
- gzip-compression B-5
- index-directory-enabled B-6
- index-directory-sort-by B-6
- langtag-revision B-7
- minimum-native-file-size B-7
- native-io-enabled B-7
- optimistic-serialization B-7
- prefer-application-packages B-8
- prefer-application-resources B-8
- prefer-forward-query-string B-9
- prefer-web-inf-classes B-9
- redirect-with-absolute-url B-9
- referer-validation B-9
- relogin-enabled B-10
- require-admin-traffic B-10
- resource-reload-check-secs B-11
- save-sessions-enabled B-11
- servlet-reload-check-secs B-11
- session-monitoring-enabled B-11
- show-archived-real-path-enabled B-11
- single-threaded-servlet-pool-size B-12

Copyright © 2007, 2026, Oracle and/or its affiliates.

temp-dir B-12
context-root B-12
description B-13
ejb-reference-description B-13
fast-swap B-13
jsp-descriptor B-13
library-ref B-16
logging B-16
ready-registration B-18
resource-description B-19
resource-env-description B-19
run-as-role-assignment B-19
security-permission B-20
security-role-assignment B-20
service-reference-description B-21
servlet-descriptor B-22
session-descriptor B-22
url-match-map B-28
virtual-directory-mapping B-29
weblogic-version B-29
wl-dispatch-policy B-30
work-manager B-30
Backward Compatibility Flags B-31
Compatibility with JSP 2.0 Web Applications B-32
JSP Behavior and Buffer Suffix B-32
Implicit Servlet 2.5 Package Imports B-32
Web Container Global Configuration B-32

C Support for GlassFish Deployment Descriptors

D Web Application Best Practices

CGI Best Practices D-1
Servlet Best Practices D-1
Best Practice When Subclassing ServletResponseWrapper D-2

E HTTP Proxy Servlet Parameters

Debug E-1
DebugConfigInfo E-1
DefaultFileName E-2

Copyright © 2007, 2026, Oracle and/or its affiliates.

FileCaching E-2
KeepAliveEnabled E-3
KeepAliveSecs E-3
MaxPostSize E-3
PathPrepend E-3
PathTrim E-4
SecureProxy E-4
WebLogicHost E-5
WebLogicPort E-5
WLCookieName E-5
WLIOTimeoutSecs E-5
WLLogFile E-6
WLProxyPassThrough E-6
WLProxySSL E-6
WLProxySSLPassThrough E-6

Copyright © 2007, 2026, Oracle and/or its affiliates.

Preface

This document is a resource for software developers who develop Web applications and components such as HTTP servlets and Jakarta Server Pages (JSPs) for deployment on WebLogic Server.

Audience

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.

It is assumed that the reader is familiar with Jakarta EE 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 .

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 for Oracle WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.
- *Upgrading Oracle WebLogic Server* contains information about Web applications, JSP, and servlet compatibility with previous WebLogic Server releases.
- For the Jakarta EE 11 platform specification and API documentation, see Jakarta EE Platform 11.

Samples and Tutorials

Oracle provides a variety of code examples and tutorials that show WebLogic Server configuration and API use, and provide practical instructions on how to perform key development tasks. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

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

New and Changed WebLogic Server Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see What's New in Oracle WebLogic Server.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which user supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that user enter.

Copyright © 2007, 2026, Oracle and/or its affiliates.

1

Understanding Web Applications, Servlets, and JSPs

Learn about WebLogic Server Web applications, servlets, and Jakarta Server Pages (JSPs). For request limits, redirect protection, and HTTP Strict Transport Security, see Web Application Security and HTTP Hardening in Chapter 2.

The Web Applications Container

A Web application contains an application's resources, such as servlets, Jakarta Server 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 Jakarta Enterprise Beans (EJBs).

Web Applications and Jakarta EE

The Jakarta EE programming model employs metadata annotations which simplify the application development process by allowing a developer to specify within the Java class itself how the application component behaves in the container, requests for dependency injection, and so on. Annotations are an alternative to deployment descriptors that were required by older versions of enterprise applications (Java EE 1.4 and earlier).

With Jakarta EE annotations, the standard `application.xml` and `web.xml` deployment descriptors are optional. The Jakarta EE programming model uses the JDK annotations feature for Web containers, such as EJBs, servlets, Web applications, and JSPs.

However, Web applications deployed on WebLogic Server can still use a standard Jakarta EE deployment descriptor file and a WebLogic-specific deployment descriptor file to define their resources and operating attributes.

Web Application Development Key Points

JSPs and HTTP servlets can access all services and APIs available in WebLogic Server. These services include EJBs, database connections by way of JDBC, JMS services, 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 can contain static resources, servlets, JSP pages, application classes, libraries, and optional deployment descriptors. The exact contents depend on the application.

A typical dynamic Web application can contain the following:

- One servlet or Jakarta Server Page (JSP), along with any helper classes.
- An optional `web.xml` deployment descriptor, which is a Jakarta EE standard XML document that describes the contents of a WAR file.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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.

Oracle recommends that you package and deploy your standalone Web applications as part of an enterprise application. This is an Oracle 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 such. For more information on archived format, see Web Application Developer Tools.

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, EJBs, messaging APIs, HTTP sessions, and other facilities of WebLogic Server.

Servlets and Jakarta EE

WebLogic Server fully supports HTTP servlets as defined in the Jakarta Servlet 6.1 specification at `https://jakarta.ee/specifications/servlet/6.1/`. HTTP servlets form an integral part of the Jakarta EE standard.

With Jakarta EE annotations, deployment descriptors are optional for many applications. Use `META-INF/application.xml` for EAR-level configuration. Use `WEB-INF/web.xml` for Web-module configuration that is not supplied by annotations or defaults. Use `WEB-INF/weblogic.xml` only when the Web module requires WebLogic Server-specific configuration. See WebLogic Annotation for Web Components.

The servlet specification defines the implementation of the servlet API and the method by which servlets are deployed in enterprise applications. Deploying servlets on a Jakarta EE-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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 failover between server down time and session sharing between clustered servers. For more information see Session Tracking from a Servlet.
  - JDBC drivers—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.
  - Jakarta Enterprise Beans—Servlets can use Jakarta Enterprise Beans (EJBs) to encapsulate sessions, data from databases, and other functionality. See Referencing External EJBs, More about the ejb-ref* Elements, and Referencing Application-Scoped EJBs.
  - Jakarta Messaging Service (JMS)—JMS allows your servlets to exchange messages with other servlets and Java programs. See *Developing JMS Applications for Oracle WebLogic Server*.
  - Java SE APIs—Servlets can use the standard Java SE APIs.
  - Forwarding requests—Servlets can forward a request to another servlet or other resource. Forwarding a Request.
- Easily deploy servlets written for any Jakarta EE-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 Jakarta API, `jakarta.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, Jakarta Server Pages (JSPs), static HTML pages, images, and security.

Jakarta Server Pages

Jakarta Server Pages (JSPs) are defined by a 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

WebLogic Server supports the Jakarta Server Pages 4.0 specification. JSP pages combine template content with Expression Language and tag libraries and are compiled by the container into servlet classes. The `weblogic.appc` compiler can precompile JSP pages and validate deployment descriptors before deployment. Application classes belong in `WEB-INF/classes`, and application libraries belong in `WEB-INF/lib`.

> **Note**
>
> JSP remains supported and is the required built-in view technology for Jakarta MVC 3.0. The compatibility warning applies to scriptlets and older proprietary frameworks, not to JSP as a whole.

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.

For new JSP pages, use Expression Language and Jakarta Tags/JSTL for presentation. Keep Java logic in controllers, servlets, or managed application components. JSP declarations, scriptlets, and expression elements remain available for existing applications but are not the recommended style for new pages.

JSPs and Jakarta EE

WebLogic JSP supports the Jakarta Server Pages specification at `https://jakarta.ee/specifications/pages/4.0/`.

- An expression language (EL) syntax that allows deferred evaluation of expressions, enables using expressions to both get and set data and to invoke methods, and facilitates customizing the resolution of a variable or property referenced by an expression.
- Support for resource injection through annotations to simplify configuring access to resources and environment data.
- Complete alignment of JSF technology tags and JSP software code.

Because JSPs are part of the Jakarta EE 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 template markup, Expression Language, and tag libraries to render dynamic Web content.
- 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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 translates the JSP into an HTTP servlet class that implements `jakarta.servlet.jsp.HttpJspPage`. When no compiled version is available, compilation is required before the page can handle a request. The `page-check-seconds` setting controls checks for source changes. Its default is 1 second in development mode and `-1` in production mode; `-1` disables automatic source-change checks.
4. WebLogic Server invokes the generated JSP servlet class to handle the 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.

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 JSP servlet class as an HTTP servlet. See Servlets.

Web Application Developer Tools

Oracle provides Oracle JDeveloper to help simplify creating, testing, debugging, and deploying Web applications that use servlets, JSP, or Jakarta Faces.

Oracle JDeveloper is an enterprise IDE providing a unified development experience for Oracle Fusion Middleware products.

Oracle JDeveloper provides advanced code editor features, collaborative teamwork development, visual development and debugging, and streamlined deployment capabilities.

Other Tools

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 or an XML editor to edit the deployment descriptors and complete the configuration of your Web application.

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 run time, your user name 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 for the Web application.

At run time, 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 Resources Using Roles and Policies for Oracle WebLogic Server*. For more

Copyright © 2007, 2026, Oracle and/or its affiliates.

information on WebLogic security, refer to *Developing Applications with the WebLogic Security Service*.

Limiting the Number of Parameters in an HTTP Request

You can prevent overloading the WebLogic Server domain with excessive parameters in HTTP requests by setting the `MaxRequestParameterCount` attribute on the `WebServer` MBean. This attribute limits the number of parameters allowed in a request. The default value of `MaxRequestParameterCount` is 10,000. If the number of parameters on an incoming HTTP request exceeds the maximum value set in the `MaxRequestParameterCount` attribute, then the following error is logged:

```
<Error> <ServletContext> <BEA-000000> <Rejecting request since max request parameter
limit exceeded 10000>
```

You can set this parameter either on the `WebServer` MBean or on the `VirtualHost` MBean. Use WLST online to set this attribute as shown in the following examples:

- Using the `WebServer` MBean

  ```
  connect('<admin_user>','<admin_pwd>','<admin_url>')
  edit()
  startEdit()
  cd('Servers/<server_name>')
  cmo.getWebServer().setMaxRequestParameterCount(1000)
  save()
  activate()
  exit()
  ```

- Using the `VirtualHost` MBean

  ```
  connect('<admin_user>','<admin_pwd>','<admin_url>')
  edit()
  startEdit()
  cd('VirtualHosts/<virtual_host>')
  cmo.setMaxRequestParameterCount(1000)
  save()
  activate()
  exit()
  ```

> **Note**
>
> If you have set `MaxRequestParameterCount` on the `WebAppContainer` MBean, Oracle recommends setting the attribute on the `WebServer` MBean instead.

Avoiding Redirection Attacks

When a request on a Web application is redirected to another location, the Host header contained in the request is used by default in the Location header that is generated for the response. Because the Host header can be spoofed—that is, corrupted to contain a different host name and other parameters—this behavior can be exploited to launch a redirection attack on a third party.

To prevent the likelihood of this occurrence, set the `FrontendHost` attribute on either the WebServerMBean or ClusterMBean to specify the host to which all redirected URLs are sent.

Copyright © 2007, 2026, Oracle and/or its affiliates.

The host specified in the `FrontendHost` attribute will be used in the Location header of the response instead of the one contained in the original request.

See `FrontendHost` in *MBean Reference for Oracle WebLogic Server*.

Using HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) is a web security policy mechanism that allows a web server to be configured so that web browsers, or other user agents, can access the server using only secure connections, such as HTTPS. Web servers declare this policy using the Strict-Transport-Security HTTP response header field.

The HSTS policy Strict-Transport-Security HTTP response header directs browsers to communicate with the web server only over secure transport such as TLS/SSL, for a specified expiration time. The HSTS policy may also specify whether the policy applies to subdomains of the host's domain name. When a browser receives and processes the HSTS header, it remembers the web server and automatically uses HTTPS for all future access to the server. Any attempts to access the web server using HTTP are automatically converted to HTTPS requests instead.

> **Note**
>
> If an application uses a mix of both HTTP and HTTPS, or some resources in an application can *only* be accessed using HTTP, then the application will be broken after you enable HSTS on WebLogic Server. To ensure that your applications continue to work after enabling HSTS, ensure that all pages in the application can be accessed using HTTPS. If any pages are hard-coded to be accessible only using HTTP, then they should be updated to be accessible using HTTPS.

WebLogic Server provides system properties to enable HSTS, and to customize the response header:

- `-Dweblogic.http.headers.enableHSTS={true|false}` - enables HSTS. The default is `false`.
- `-Dweblogic.http.headers.hsts.maxage=`*`max-age-seconds`* - sets the policy expiration time. The default is `31536000` seconds (one year).
- `-Dweblogic.http.headers.hsts.includesubdomains={true|false}` - specifies whether the HSTS policy applies to the subdomains of the host domain. The default is `true`.
- `-Dweblogic.http.headers.hsts.preload={true|false}` - specifies whether the domain is requesting inclusion in the HSTS preload list maintained by Google. The default is `true`. All sites approved for inclusion are hardcoded into this list and can only be accessed using HTTPS in Chrome and other browsers.

For details about these system properties, see HTTP Strict Transport Security in *Command Reference for Oracle WebLogic Server*.

For more information about HSTS, see the following documents:

- HTTP Strict Transport Security (HSTS) standard at `https://www.rfc-editor.org/rfc/rfc6797`
- Strict-Transport-Security on MDN Web Docs at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security`

Copyright © 2007, 2026, Oracle and/or its affiliates.

2

Creating and Configuring Web Applications

Learn how to create and configure WebLogic Web applications.
This chapter includes the following sections:

WebLogic Web Applications and Jakarta EE

This chapter describes the WebLogic Server configuration, packaging, URL, virtual-host, library, and compression settings for a Web module. For WAR structure and introductory Jakarta EE concepts, see Chapter 1.

With Jakarta EE annotations, the standard `application.xml` and `web.xml` deployment descriptors are optional. The Jakarta EE programming model uses the JDK annotations feature for Web containers, such as EJBs, servlets, Web applications, and JSPs. See WebLogic Annotation for Web Components.

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

Directory Structure

Web applications use a standard directory structure defined in the Jakarta EE 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. Oracle recommends that you package and deploy your exploded Web application as part of an enterprise application. This is an Oracle 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.

Place tag files under `WEB-INF/tags`. Place standalone Tag Library Descriptor (TLD) files under `WEB-INF`. A packaged tag-library JAR belongs in `WEB-INF/lib` and contains its TLD under `META-INF`. Application tag-handler classes that are not packaged in a JAR belong under `WEB-INF/classes`.

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

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 JDBC, JMS services, XML, and more.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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:

Example 2-1 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

Learn how to create a Web application as part of an enterprise application using the split development directory structure.

See Creating a Split Development Directory Environment, Building Applications In a Split Development Directory, and Deploying and Packaging From a Split Development Directory in *Developing Applications for Oracle WebLogic Server*.

You may want to use developer tools included with WebLogic Server for creating and configuring Web applications. See Web Application Developer Tools.

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, execute the `setWLSEnv.cmd` command, located in the directory *`WL_HOME`*`\server\bin\`, where *`WL_HOME`* is the top-level directory in which WebLogic Server is installed.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

> **Note**
>
> On UNIX operating systems, the `setWLSEnv.sh` command does not set the environment variables in all command shells. Oracle recommends that you execute this command using the Korn shell or bash shell.

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 for Oracle WebLogic Server*.
   b. Edit the deployment descriptors as needed to fine-tune the behavior of your enterprise application. See Web Application Developer Tools.
   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 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.
   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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Step Four: Execute the Split Development Directory Structure Ant Tasks

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

1. Execute the `wlcompile` Ant task to invoke the `javac` compiler. This compiles your Web application Java components into an output directory: `/build/myEAR/myWebApp/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/WEB-INF/classes`.

Configuring How a Client Accesses a Web Application

You construct the URL that a client uses to access a Web application using a specific pattern.

```
https://host:port/ContextPath/servletPath/pathInfo
```

Where

- `hoststring` is either a host name that is mapped to a virtual host or `hostname:portNumber`.
- `ContextPath` is the context root assigned to the deployed Web application. It is not necessarily the archive name or application name.
- `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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Remote 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="virtualhost-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 a certain order:

Order of loading:

1. Context listeners
2. Filters
3. Servlets

Order of destruction:

1. Servlets
2. Filters
3. Context listeners

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta EE Web Application Libraries

A Jakarta EE Web application library is a standalone Web application module registered with the Jakarta EE application container upon deployment. With WebLogic Server, 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 Jakarta EE libraries and their deployment, see Creating Shared Jakarta EE Libraries and Optional Packages in *Developing Applications for Oracle WebLogic Server*.

Enabling GZIP Compression for Web Applications

The WebLogic Server Web container supports HTTP content-encoding GZIP compression, which is part of HTTP/1.1. With GZIP compression, you can reduce the size of the data that a Web browser has to download, improving network bandwidth.

For general information about content-encoding and GZIP compression, see the Hypertext Transfer Protocol HTTP/1.1 Specification.

You can enable and configure content-encoding GZIP compression at the domain level or Web application level.

To configure domain-wide GZIP compression, use WLST or the WebLogic Remote Console to set the following attributes under **WebAppContainer > Gzip Compression**. The corresponding runtime configuration is exposed by `GzipCompressionMBean`.

Table 2-1 Domain-Level GZIP Compression Attributes

Attribute	Description	Default Value
`GzipCompressionEnabled`	Enables GZIP compression for all Web applications in the domain.	`false`

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Domain-Level GZIP Compression Attributes

Attribute	Description	Default Value
`GzipCompressionMinContentLength`	Specifies the minimum file size to trigger compression in Web applications. This attribute allows you to bypass small-sized resources where compression would not yield a great return but use unnecessary CPU.	`2048`
`GzipCompressionContentType`	Specifies the content types to include in compression.	`"text/html, text/xml, text/plain"`

To configure GZIP compression for a specific Web application, use the gzip-compression element in the weblogic.xml deployment descriptor container-descriptor element. See gzip-compression.

Application-level values override domain-level values. Therefore, any `gzip-compression` values set in `weblogic.xml` take precedence over domain-wide values set in the `GzipCompressionMBean` or default values.

WebLogic Server selects GZIP settings in this order:

1. Values in the Web application's `WEB-INF/weblogic.xml` take precedence.
2. If the application does not set a value, the domain-level GZIP setting is used.
3. If neither the application nor the domain sets a value, then WebLogic Server uses the documented default.

To write HTTP compression diagnostics to the server log, start the server with `-Dweblogic.debug.DebugHttp=true`. Enabling HTTP debug logging can generate substantial diagnostic output; use it for troubleshooting rather than as a normal production setting.

Copyright © 2007, 2026, Oracle and/or its affiliates.

3

Using Jakarta MVC 3.0 in WebLogic Server Web Applications

Jakarta MVC 3.0 is the latest major version of the Jakarta Model-View-Controller (MVC) specification, part of the Jakarta EE platform.
Jakarta MVC 3.0 defines a standard way to build server-side web applications using the classic MVC architectural pattern. Jakarta MVC 3.0 is aligned with Jakarta EE 11 and Jakarta RESTful Web Services 4.0

The specification is based on Jakarta RESTful Web Services (JAX-RS), meaning controllers are implemented as JAX-RS resource classes, while views are typically rendered using technologies such as JSP or Facelets.

> **Note**
>
> Jakarta MVC 3.0 requires a built-in JSP view engine. An implementation can support additional view engines, but Jakarta MVC 3.0 does not require Facelets support. Facelets is the standard view technology for Jakarta Faces and normally uses `.xhtml` files.

For more information about the Jakarta MVC 3.0 architecture, see the Jakarta MVC 3.0 specification.

Developing Jakarta MVC Applications

Learn how to develop, configure, secure, and deploy Jakarta Model View Controller (MVC) 3.0 Web applications on Oracle WebLogic Server 26.
Jakarta MVC is a specification, not a complete application recipe. It defines the portable contracts for actions, models, views, binding, security, events, application configuration, view engines, locale resolution, and form-method override. Oracle WebLogic Server supplies the Jakarta MVC implementation and the Jakarta EE services on which it runs. This chapter connects those two perspectives: it explains the specification contract, then shows how a Web application packages and uses that contract on Oracle WebLogic Server.

The examples in this chapter use a small catalog application throughout. The application lists products, displays an edit form, validates submitted data, redirects after a successful update, and renders JSP views. Reusing one application matters: it lets you see how a model created by a controller becomes view data, how a form reaches that controller, and where cross-cutting behavior belongs.

Before You Begin

The examples assume you are using the following:

- Oracle WebLogic Server 26
- Jakarta EE 11 APIs
- a WAR application

Copyright © 2007, 2026, Oracle and/or its affiliates.

They use JSP as the view technology because JSP is the built-in view technology required by Jakarta MVC. An EAR can contain the same Web module when the application also needs enterprise modules.

Compile against the APIs supplied by the target server. The application owns its controllers, CDI beans, JSPs, descriptors, and business libraries; it does not need to bundle an MVC implementation to use Jakarta MVC.

```
<dependencies>
  <dependency>
    <groupId>jakarta.platform</groupId>
    <artifactId>jakarta.jakartaee-web-api</artifactId>
    <version>11.0.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>jakarta.mvc</groupId>
    <artifactId>jakarta.mvc-api</artifactId>
    <version>3.0.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Understanding Jakarta MVC Applications

This topic describes the Jakarta MVC applications, their relationship to Jakarta EE Web applications, and the Oracle WebLogic Server environment in which they run.
This topic includes the following sections:

- Understanding the Jakarta MVC Web Application Model
- Following an MVC Request Through the Application
- Using the Jakarta MVC APIs and Jakarta EE Technologies
- Using Jakarta MVC on Oracle WebLogic Server
- Planning the Application Archive

Understanding the Jakarta MVC Web Application Model

Jakarta MVC is an action-based Web programming model. A browser requests an action; a controller obtains input and invokes application logic; the controller supplies a model and selects a view; the view creates the HTML that is returned to the browser. The model is deliberately the connection between controller and view. A controller should decide what data a user can see; a JSP should decide how that data is presented.

This is different from a conventional Jakarta RESTful Web Services endpoint. A REST resource commonly returns a representation such as JSON. An MVC controller normally returns a view name. The same Jakarta RESTful Web Services routing annotations select both kinds of method, but `@Controller` tells the MVC runtime to treat the action result as MVC behavior.

Copyright © 2007, 2026, Oracle and/or its affiliates.

The catalog application has both kinds of resource in one Web module. Its HTML action is a controller, whereas an optional stock API returns JSON. Keeping their URLs separate makes the intent visible to clients and to maintainers.

```
/catalog/mvc/products        MVC action -> JSP page
/catalog/api/products/42     REST resource -> JSON representation
```

Following an MVC Request Through the Application

Consider a user who opens the catalog page. The request has to pass through several standard Jakarta EE layers before a JSP can use `${products}`. Seeing that path first makes the later chapters on binding, CSRF, providers, and view engines easier to place.

1. The Web application receives a request below its context root.
2. The Jakarta RESTful Web Services application path and resource annotations select a controller action.
3. CDI supplies the controller and its injected services for the request.
4. The controller reads input, invokes the application service, and adds view data to `Models` or a named CDI bean.
5. Jakarta MVC selects a view engine for the returned view name. For a JSP view, the engine forwards to the view and exposes model entries as request attributes.
6. JSP and Expression Language render the response. The important boundary for application code is step 4. It should contain the application decision: whether the caller is authorized, whether the input is valid, which service operation to perform, and which view or redirect follows. The framework performs routing, contextual injection, view selection, and JSP dispatch according to its Jakarta EE and Jakarta MVC contracts.

Using the Jakarta MVC APIs and Jakarta EE Technologies

Jakarta MVC is composed from existing Jakarta EE technologies. Use their standard APIs instead of trying to reproduce framework behavior in a controller.

For example, a `ProductController` is not a special proprietary controller class. It is a CDI-managed Jakarta RESTful Web Services resource with the portable `@Controller` annotation. That is why it can inject CDI services and use familiar route annotations.

Technology	Role in an MVC application	Typical application use
Jakarta RESTful Web Services	Matches URL, HTTP method, and request parameters.	`@ApplicationPath`, `@Path`, `@GET`, `@POST`, parameter annotations.
CDI	Creates controllers and application beans.	`@RequestScoped`, injection, named model beans, lifecycle observers.
Jakarta MVC	Connects actions, models, views, binding, CSRF, and URI generation.	`@Controller`, `Models`, `MvcContext`, `BindingResult`.
Jakarta Servlet	Defines the WAR request, session, filter, and deployment model.	`web.xml`, sessions, application filters where needed.
JSP and Expression Language	Renders the selected HTML view.	JSP, tag libraries, `${mvc...}`, escaped output.
Jakarta Bean Validation	Validates submitted form data.	`@Valid`, constraint annotations on a form bean.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using Jakarta MVC on Oracle WebLogic Server

Oracle WebLogic Server 26 supplies Jakarta MVC 3.0. The default MVC implementation is Eclipse Krazo, which works with the server's Jakarta RESTful Web Services, CDI, Servlet, and JSP facilities. An application uses the Jakarta APIs; it does not need to call Eclipse Krazo implementation APIs to create a controller or render a JSP. The server-specific part of an MVC application is therefore mostly ordinary WebLogic Server Web-application work: package a WAR, choose a context root, deploy it to a target, and use server-provided APIs at run time. The MVC programming model remains the Jakarta MVC model. This guide calls out a WebLogic Server concern only when it affects a deployment decision, such as the server-supplied runtime or a `weblogic.xml` context root.

Planning the Application Archive

Put application code and views in the Web module. This layout shows the catalog application before it has grown into a larger service:

```
catalog.war
├── WEB-INF/
│   ├── beans.xml
│   ├── classes/com/example/catalog/
│   │   ├── CatalogApplication.class
│   │   ├── ProductController.class
│   │   └── ProductService.class
│   ├── lib/                          application-owned libraries
│   ├── views/products/list.jsp
│   ├── web.xml                        optional standard configuration
│   └── weblogic.xml                   optional WebLogic Server configuration
└── css/ and images/                   static resources
```

Put views below `WEB-INF/views` when using the default JSP view folder. A browser cannot retrieve content below WEB-INF directly. That structural protection means that the controller remains the entry point that supplies the model and applies application authorization. `beans.xml` is not an MVC switch. It is a CDI bean-archive descriptor. Include it when the application needs to select CDI discovery behavior or make that behavior explicit. Similarly, add `weblogic.xml` only for a WebLogic Server setting the application actually requires, such as a fixed context root.

Important Development and Security Considerations

Keep the following decisions visible in the application design:

- Give the Jakarta RESTful Web Services application a nonempty application path, such as `mvc`, and keep page and API routes distinct.
- Use request-scoped controllers unless a different CDI lifecycle is needed and understood.
- Keep controller actions small: obtain input, call an application service, prepare view data, then choose a view or redirect.
- Treat models as data for the selected view. Do not use a model as a substitute for a user session or a service cache.
- Escape untrusted values in the output context in which they are rendered.
- Protect state-changing actions with Web application authorization and Jakarta MVC CSRF protection; hiding a link or a JSP is not authorization.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Test the deployed URL, not only a controller unit test. The deployment test verifies the application path, CDI discovery, view location, and JSP rendering together.

The next topic, Using Models, Views, and Controllers turns this model into controllers, model data, views, URI generation, and redirects.

Using Models, Views, and Controllers

This topic describes how to implement the central Jakarta MVC contract: controllers select actions, models carry presentation data, and views render the result.
This topic includes the following sections:

- Choosing Controller Actions
- Selecting a View or Response
- Preparing View Data
- Rendering JSP Views
- Building Application URIs
- Redirecting After an Action

Choosing Controller Actions

An MVC controller is a Jakarta RESTful Web Services resource method annotated with `@Controller`. Put `@Controller` on a class when all of its resource methods return HTML views. Put it on individual methods when the class also contains a conventional REST endpoint. The annotation is the bridge between REST routing and MVC view processing; it does not replace `@Path` or an HTTP method annotation.

The catalog list action illustrates the division of responsibility. Routing selects `list`; the controller asks the service for products; `Models` makes the result available to the view; the returned string names the view.

```
@Controller
@RequestScoped
@Path("products")
public class ProductController {
    @Inject ProductService productService;
    @Inject Models models;

    @GET
    public String list() {
        models.put("products", productService.findAll());
        return "products/list.jsp";
    }

    @GET
    @Path("{id}")
    public String detail(@PathParam("id") long id) {
        models.put("product", productService.find(id));
        return "products/detail.jsp";
    }
}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

Use a CDI scope for a controller. `@RequestScoped` suits the common case: the controller represents one request and does not retain mutable state after the response. CDI injection, interceptors, and decorators are then available in the normal way. Do not create a controller with `new`; that bypasses CDI and creates a different lifecycle from the one the application declares.

Selecting a View or Response

Jakarta MVC defines three common ways for an action to select a view. Choose the form that communicates the action's intent rather than treating every action as a string-returning method.

Action result	Use it when	Example
`String`	The view is chosen at run time.	Return "`products/detail.jsp`".
`void` with `@View`	The action always has one view.	A help page with no decision to make.
`Response`	Status, headers, media type, or redirect metadata matter.	A `303 See Other` after an update.

`@View` also provides the default view if a non-void action returns `null`. That makes an explicit fallback readable when a controller has a small conditional decision:

```
@GET
@Path("help")
@View("help.jsp")
public void help() {
}

@GET
@Path("account")
@View("account-not-found.jsp")
public String account(@QueryParam("id") String id) {
    return accountService.exists(id) ? "account.jsp" : null;
}
```

Returning a `Response` does not turn an MVC action into an API endpoint. Its status and headers remain useful when the entity is a view name. Use it, for example, when a page must explicitly prevent caching. Use an ordinary non-controller resource method when the response body itself is JSON or another representation.

Preparing View Data

`Models` is a request-oriented map for data that a controller has selected for the next view. The controller and JSP agree on a name such as `products`; the JSP does not need to know how the service obtained the list. Before the JSP is rendered, MVC exposes model entries as request attributes, so `${products}` is available to Expression Language.

Use a named CDI bean when presentation data has a useful type and lifecycle. For example, a multi-step display may be clearer as `CatalogModel` than a set of unrelated string keys. The decision is an application design choice; both approaches remain view data, not a replacement for persistent user state.

```
@Named("catalog")
@RequestScoped
public class CatalogModel {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    private List<Product> products;
    public List<Product> getProducts() { return products; }
    public void setProducts(List<Product> products) { this.products =
products; }
}
```

Do not place secrets, authorization decisions, or unfiltered user input in a model merely because a JSP can read it. Select the minimal data required by the authorized view and encode it correctly when it is displayed.

Rendering JSP Views

JSP is the built-in Jakarta MVC view technology. With the default view folder, the relative name `products/list.jsp` identifies `WEB-INF/views/products/list.jsp`. The controller returns a logical view name; the JSP handles HTML markup and iteration.

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<h1>Products</h1>
<ul>
  <c:forEach var="product" items="${products}">
    <li><c:out value="${product.name}" /></li>
  </c:forEach>
</ul>
```

`<c:out>` is appropriate for text in an HTML element. Other output contexts need their own encoding rules: an HTML attribute, a URL, JavaScript, and CSS are not interchangeable. Keep database access, authorization checks, and state changes out of a JSP so the same controller rules apply regardless of which view is selected.

Building Application URIs

Hard-coding context roots and controller paths in JSPs makes a deployment move needlessly fragile. `MvcContext`, exposed in a JSP as `mvc`, can build a URI from a controller method. Give routes that are referenced from views a stable `@UriRef` so a Java class rename does not silently change the view contract.

```
@GET
@Path("{id}")
@UriRef("product-detail")
public String detail(@PathParam("id") long id) {
    // prepare the model
    return "products/detail.jsp";
}
```

```
<a href="${mvc.uri('product-detail', {'id': product.id})}">
  <c:out value="${product.name}" />
</a>
```

The parameter-map keys must match the route's path, query, or matrix parameter names. Test generated links after changing a route, especially when the application is deployed under a non-default context root or fronted by a proxy.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Redirecting After an Action

A successful form POST should normally redirect to a GET page. This POST-redirect-GET pattern prevents a browser refresh from repeating the update and gives the resulting page its own address. Return `redirect:` when a simple redirect is enough, or return `Response.seeOther` when the response metadata is part of the action's intent.

```
@Inject
private BindingResult bindingResult;
@POST
public String create(@BeanParam @Valid ProductForm form) {
    if (bindingResult.isFailed()) {
        return "products/edit.jsp";
    }
    Product product = productService.create(form);
    return "redirect:products/" + product.getId();
}
```

`@RedirectScoped` is for the narrow case where a CDI bean must survive the redirect and the immediately following request, such as a one-time success message. It is not a general session scope. Test the complete browser flow: submit the form, observe the redirect status and Location header, follow the GET, then refresh the GET to confirm that the original operation is not run again.

Using Data Binding in Jakarta MVC Applications

Learn how to turn browser input into application values, report conversion and validation failures, and redisplay a form without losing your work.
This topic includes the following sections:

- Understanding the Form Processing Contract
- Designing a Form Bean
- Retaining Binding Failures in the Controller
- Rendering Field Errors
- Converting Values Reliably
- Testing a Form Action

Understanding the Form Processing Contract

An HTML form contains strings. An application action usually needs typed values: a price, quantity, date, identifier, or domain object. Jakarta RESTful Web Services performs parameter binding and Jakarta Bean Validation can validate the resulting object. Without MVC binding support, a conversion or validation failure can prevent the resource method from running, which is a reasonable behavior for an API but not a helpful browser form experience.

`@MvcBinding` changes that experience. It tells Jakarta MVC to invoke the controller and expose the failure through `BindingResult`, allowing the action to return the same JSP with field values and messages. The controller still must check the result before it calls a state-changing service.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Designing a Form Bean

Group related form fields in a `@BeanParam` class. Put syntactic rules on the form at the boundary—required fields, range, length, and format—not in the JSP. The service then receives a form that has passed the controller's validation decision.

```
public class ProductForm {
    @FormParam("name")
    @MvcBinding
    @NotBlank(message = "Enter a product name")
    @Size(max = 80)
    private String name;

    @FormParam("price")
    @MvcBinding
    @NotNull(message = "Enter a price")
    @DecimalMin(value = "0.01")
    private BigDecimal price;

    public String getName() { return name; }
    public BigDecimal getPrice() { return price; }
}
```

The form bean describes input; it need not be the same class as an entity or the object rendered by a JSP. Keeping those concepts distinct prevents clients from binding fields that should not be editable, such as an owner or an authorization flag.

Retaining Binding Failures in the Controller

Inject `BindingResult` alongside the form. A failed conversion—for example, `abc` in a `BigDecimal` field—and a violated Bean Validation constraint both appear in the binding result. The action selects the edit view before calling the service when either one occurs.

```
@Inject
private BindingResult bindingResult;
@POST
@Path("{id}")
public String update(@PathParam("id") long id,
        @BeanParam @Valid ProductForm form) {
    if (bindingResult.isFailed()) {
        models.put("productId", id);
        return "products/edit.jsp";
    }
    productService.update(id, form);
    return "redirect:products/" + id;
}
```

The WebLogic Server deployment does not require special form-binding configuration for this portable MVC pattern. The application supplies the controller, form bean, constraints, and JSP; the server-provided MVC runtime participates in the Web application's normal request processing.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Rendering Field Errors

A form view needs two kinds of information after a failed submission: the value the user attempted to enter and the reason it is unacceptable. Keep the JSP focused on displaying those values. It should not decide whether an update is allowed or attempt its own numeric conversion.

```
@Inject
private BindingResult bindingResult;
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<label for="price">Price</label>
<input id="price" name="price" value="<c:out value='${form.price}'/>" />
<c:forEach items="${bindingResult.getErrors('price')}" var="error">
  <p id="price-error" class="error"><c:out value="${error.message}" /></p>
</c:forEach>
```

Add `aria-describedby="price-error"` to the price input when the error element is rendered. `BindingResult` has `isFailed()` without a parameter, `getErrors(String)`, and `getAllMessages()` without a parameter.

Use the binding-result API supported by the selected view technology to render messages. Give error text an accessible association with the corresponding input, and preserve only safe, expected values. Passwords and other secrets should not be copied back into a page merely to repopulate a form.

Converting Values Reliably

Use typed form properties where a field has a clear type. MVC conversion covers primitives, wrapper types, numeric values, booleans, and other types for which an applicable converter exists. Locale can influence human-entered numeric and date values; use the MVC request locale consistently for parsing, formatting, and error messages.

For a domain-specific value, make the conversion rule explicit and test it. Do not accept an arbitrary identifier just because it can be parsed. A SKU, currency code, or account identifier may need an allowlist or a service lookup after conversion succeeds.

Testing a Form Action

Test the deployed form as a request sequence, not just as a Java method. At a minimum, verify a valid submission, a missing required value, an invalid typed value, an out-of-range value, and a failure that must not call the service. For each invalid case, assert that the response renders the expected form and field message. For the valid case, assert the redirect and follow it to the GET view. Add CSRF and authorization cases from the security chapter, because they protect the same state-changing action.

Securing Jakarta MVC Applications

Learn how Web application security, Jakarta MVC CSRF protection, and output encoding work together to protect browser-facing actions on Oracle WebLogic Server.
This chapter includes the following sections:

- Securing the Action, Not Just the Page
- Using Implicit CSRF Protection

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Selecting Explicit CSRF Protection
- Encoding View Output
- Handling Security Failures
- Verifying a Protected Form

Securing the Action, Not Just the Page

An MVC application is a Web application. Apply Oracle WebLogic Server Web application authentication and authorization to the URLs that perform actions, then apply business authorization in the service layer when an operation also depends on ownership or other domain data. A hidden button, an unlinked JSP, or a disabled browser field is not an authorization rule; a client can still send the request.

Treat each state-changing controller action as the security boundary. The catalog update action, for example, needs an authorized caller, a valid CSRF token, valid form data, and an application-level decision that the caller can change that product. These checks solve different problems and none replaces another.

Using Implicit CSRF Protection

Jakarta MVC 3.0 uses implicit CSRF protection by default. In implicit mode, the MVC runtime protects controller actions as defined by the specification. The view obtains the token field name and value from the MVC Expression Language object and submits them with the form.

```
<form method="post" action="${mvc.uri('product-update', {'id': product.id})}">
  <input type="hidden" name="${mvc.csrf.name}" value="${mvc.csrf.token}" />
  <label for="name">Name</label>
  <input id="name" name="name" value="<c:out value='${product.name}'/>" />
  <button type="submit">Save</button>
</form>
```

The token belongs in every application form that submits a protected action, including forms that use method override. Generate it in the view instead of inventing a field name or copying a value between users. If a JavaScript client submits a request rather than a form, configure and use the MVC CSRF header name consistently with that client.

Selecting Explicit CSRF Protection

Use the standard CSRF property to select explicit protection only when the application has a clear reason to identify protected actions individually. In that mode, mark the relevant controller action with `@CsrfProtected`.

```
@POST
@Path("{id}")
@CsrfProtected
@UriRef("product-update")
public String update(@PathParam("id") long id,
        @BeanParam @Valid ProductForm form) {
    // Check binding, authorize the operation, update, and redirect.
    return "redirect:products/" + id;
}
```

Explicit mode trades the broad default for a maintenance obligation: every new state-changing action must be reviewed and marked. Document why that tradeoff is appropriate and add a

Copyright © 2007, 2026, Oracle and/or its affiliates.

security test that fails if an unmarked action is introduced. Do not disable CSRF protection merely because an action also uses authentication or role constraints.

Encoding View Output

CSRF protects a request from being forged; it does not make returned content safe to render. Encode values for their output context. In a JSP HTML text node, use `<c:out>` or an equivalent HTML-text encoder. For an attribute, URL, JavaScript, or CSS context, use the encoder appropriate to that context.

```
<p><c:out value="${product.description}" /></p>
```

MVC also exposes encoders through `MvcContext` for views or application code that need the portable API. Keep the choice near the sink: a string encoded for an HTML element is not automatically safe in a script or URL. Do not solve output encoding by stripping arbitrary characters at input time, because the same value can be valid in one output context and unsafe in another.

Handling Security Failures

Plan the user experience for authentication, authorization, CSRF, binding, and application failures separately. A missing or invalid CSRF token is a request validation failure, not a reason to display the submitted data back to the caller. A role failure should use the Web application's configured access control behavior. An invalid form value can normally return the edit page with field messages.

Exception mappers and filters can be useful for a consistent error response, but they must not turn a security failure into a successful-looking page or leak request details. Preserve the original status and log enough server-side context for diagnosis without recording passwords, CSRF tokens, or other sensitive form values.

Verifying a Protected Form

Run a browser-level or HTTP-client test for each protected form. First request the form and retain the session state. Then verify a valid token succeeds, a missing token fails, an altered token fails, an unauthenticated caller is handled as configured, and an unauthorized authenticated caller cannot perform the action. Repeat the test for any form-method-override action. Deploy behind the production proxy path when relevant so HTTPS and session-cookie behavior are part of the evidence, not an assumption.

Using Events in Jakarta MVC Applications

Learn how Jakarta MVC lifecycle events let an application observe controller, view, and redirect processing without placing cross-cutting observation code in every controller. This topic includes the following sections:

- Understanding MVC Lifecycle Events
- Choosing an Appropriate Observer
- Observing Controller and View Processing
- Observing Redirects
- Keeping Observers Safe and Fast

Copyright © 2007, 2026, Oracle and/or its affiliates.

Understanding MVC Lifecycle Events

Jakarta MVC publishes CDI events around controller execution, view rendering, and redirects. Events are useful when the same observation applies to many actions: a request metric, audit correlation, or a diagnostic marker. They are not a second controller mechanism. The controller still owns application decisions and the selected view.

MVC events are synchronous CDI events. An observer runs on the request path, which means it can see request context but can also add latency or fail the same request if it performs expensive work.

Choosing an Appropriate Observer

Choose the event closest to the fact being observed. A before-controller event answers whether an action is about to run; an after-controller event is useful for the completed action; view events distinguish action selection from actual rendering; redirect events observe the destination selected by an action.

Keep event observers separate from authorization and business logic. If an operation must be authorized, the controller or service must enforce it even when no observer is present.

Observing Controller and View Processing

The following observer records a small metric tag after a controller action. It reads metadata and delegates only a bounded local operation. The metric system, not the observer, should handle batching or remote export.

```
@ApplicationScoped
public class MvcMetrics {
    public void afterController(@Observes AfterControllerEvent event) {
        metrics.increment("mvc.controller.completed",
            event.getResourceInfo().getResourceMethod().getName());
    }
}
```

Use the event API to obtain the information the event defines. Do not parse a view name or redirect location to reconstruct routing decisions that the controller can express directly with URI APIs.

Observing Redirects

Redirect events are useful for diagnostics and audit metadata after the controller has chosen a safe destination. Do not rewrite the destination in an observer: redirect construction belongs in the controller, where it can use trusted route references and application authorization information.

Keeping Observers Safe and Fast

Do not perform remote calls, database queries, blocking I/O, or expensive aggregation directly in an MVC event observer. Record the minimal information needed and use an application-managed asynchronous design for later work. Test that an observer failure does not hide the original controller or JSP failure, and test the relevant event sequence with an actual deployed action.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Configuring Jakarta MVC Applications

Learn how to configure a Jakarta MVC application through the standard Jakarta RESTful Web Services application model and how that configuration is deployed as a Web application on Oracle WebLogic Server.
This topic includes the following sections:

- Defining the MVC Application Boundary
- Obtaining Request MVC Context
- Configuring Portable MVC Properties
- Using Providers with MVC
- Understanding Annotation Inheritance
- Configuring the WebLogic Server Context Root

Defining the MVC Application Boundary

An MVC controller is a Jakarta RESTful Web Services resource in an application. The application class establishes the URL space in which resource paths are matched and is the portable place to set MVC properties or explicitly register application classes. A nonempty application path keeps that URL space distinct from the Web application's context root and from JSP resources.

```
@ApplicationPath("mvc")
public class CatalogApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        return Set.of(ProductController.class, AccountController.class);
    }
}
```

With a context root of `/catalog`, `@ApplicationPath("mvc")`, and `@Path("products")`, the product action is `/catalog/mvc/products`. This separation also means the view folder can stay under `WEB-INF`, outside the URLs that browser clients request.

Use the standard application mechanism that matches the rest of the Web application. An `Application` subclass is a simple choice when annotations and properties are sufficient. A `ResourceConfig`-based application can be useful for an application that already uses Jersey's programmatic registration model. In either case, application code should register its own resources and extensions through supported application APIs, rather than depend on MVC implementation bootstrap classes.

Obtaining Request MVC Context

`MvcContext` represents MVC information for the current request: the base path, locale, URI builder, CSRF helper, and encoders. Inject it into a controller when the controller needs that information; JSPs access the corresponding object as `mvc` in Expression Language.

```
@Inject MvcContext mvc;
@Inject Models models;

@GET
@Path("status")
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
public String status() {
    models.put("basePath", mvc.getBasePath());
    models.put("locale", mvc.getLocale());
    return "status.jsp";
}
```

The context is request-specific. Copy a needed value into a view model or another defined state mechanism; do not retain `MvcContext` in a session- or application-scoped object for later requests.

Configuring Portable MVC Properties

Jakarta MVC defines properties for view selection and CSRF behavior. Set them on the application class so the configuration lives next to the routes it affects. The following catalog example moves JSPs to a named view folder and keeps the specification's implicit CSRF mode while selecting a header name for browser clients that use header submission.

```
@ApplicationPath("mvc")
public class CatalogApplication extends Application {
    @Override
    public Map<String, Object> getProperties() {
        Map<String, Object> properties = new HashMap<>();
        properties.put(ViewEngine.VIEW_FOLDER, "/WEB-INF/catalog-views/");
        properties.put(Csrf.CSRF_PROTECTION, Csrf.CsrfOptions.IMPLICIT);
        properties.put(Csrf.CSRF_HEADER_NAME, "X-Catalog-CSRF");
        return properties;
    }
}
```

Changing a property changes an application-level behavior, not merely a JSP detail. Keep a custom view folder below WEB-INF; test every protected form when CSRF configuration changes; and make the configured header name part of the client-side request contract. The implementation reference lists the standard property names and defaults.

Using Providers with MVC

Jakarta MVC operates inside the Jakarta RESTful Web Services pipeline. That is how a matched controller action can be recognized as MVC, how a returned view name reaches a view engine, and how ordinary JAX-RS extensions can still take part in the same request. Eclipse Krazo, the default implementation, uses that extension pipeline internally; those implementation providers are not an application API.

The application-facing consequence is simpler. An application can use standard JAX-RS providers—filters, interceptors, exception mappers, message-body providers, and features—with MVC routes just as it can with other resources. Use a provider for a cross-cutting HTTP concern. Do not put an authorization policy, a database update, or view-selection logic into a filter merely because it runs before or after an action.

Annotate a provider with `@Provider` for provider discovery, or register the application's provider explicitly through its application configuration. Add `@Priority` only when the relative position is meaningful to the application.

```
@Provider
@Priority(Priorities.HEADER_DECORATOR)
public class CacheControlResponseFilter implements ContainerResponseFilter {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    @Override
    public void filter(ContainerRequestContext request,
            ContainerResponseContext response) {
        if (request.getUriInfo().getPath().startsWith("products/")) {
            response.getHeaders().putSingle("Cache-Control", "no-store");
            response.getHeaders().putSingle("X-Content-Type-Options",
"nosniff");
        }
    }
}
```

For request filters and read/write interceptors, lower priority numbers run first. Response filters run in reverse priority order. The order of components with the same priority is implementation-defined. Use a standard `Priorities` constant when it expresses the concern, and test the provider with the MVC action, form, redirect, or JSP response it changes. This is especially important for a provider that reads or changes headers, entities, or exceptions.

Understanding Annotation Inheritance

MVC annotations follow the Jakarta RESTful Web Services annotation-inheritance rules. A resource method can inherit relevant annotations from a superclass or interface only when the implementation method does not declare an annotation of the same kind. Superclass annotations take precedence over interface annotations.

This permits a reusable route contract, but it can hide the route or MVC behavior of a controller. Prefer an explicit annotation on an action when a reader would otherwise have to inspect several interfaces to understand its HTTP method, path, or view behavior. Test inherited routes after refactoring; annotation inheritance is a route contract, not just a Java style choice.

Configuring the WebLogic Server Context Root

The context root is the WebLogic Server deployment portion of the URL. Use the deployment configuration when its value is environment-specific. When the WAR must declare a fixed context root, use `WEB-INF/weblogic.xml`:

```
<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app">
  <context-root>/catalog</context-root>
</weblogic-web-app>
```

Do not add a WebLogic Server descriptor solely to enable MVC. MVC routes, models, views, binding, and providers are configured through their Jakarta APIs. After deployment, test the final URL with the selected context root and application path, since a successful unit test cannot verify that URL composition.

Additional Jakarta MVC Capabilities

Learn how to extend Jakarta MVC applications using view engines and other methods.
This section includes the following topics:

Using View Engines in Jakarta MVC Applications

Learn how views are selected, how JSP views use model data, and when a custom Jakarta MVC view engine is an appropriate application extension.
This topic includes the following sections:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Understanding View Selection
- Using the Built-In JSP View Engine
- Changing the View Folder Convention
- Providing a Custom View Engine
- Diagnosing a View Failure

Understanding View Selection

A controller's view name is an instruction to the MVC runtime, not a file read performed by the controller. MVC asks available `ViewEngine` beans which ones support that name, then uses the selected engine to render it. This keeps the controller independent of the template technology: the controller chooses `products/list.jsp`; the JSP engine owns JSP rendering.

> **Note**
>
> Jakarta MVC 3.0 requires a built-in JSP view engine. An implementation can support additional view engines, but Jakarta MVC 3.0 does not require Facelets support. Facelets is the standard view technology for Jakarta Faces and normally uses `.xhtml` files.

Oracle WebLogic Server supplies the required JSP view-engine support with its Jakarta MVC runtime. Use it for ordinary server-rendered HTML. A custom engine is an application extension for a genuinely different view technology, not a replacement for the server's MVC or JSP libraries.

Using the Built-In JSP View Engine

By default, a relative JSP name is resolved below `/WEB-INF/views/`. The catalog action can therefore return `products/summary.jsp`, while the archive contains `WEB-INF/views/products/summary.jsp`. The location protects the view from direct browser access and keeps the controller as the action entry point.

```
@GET
@Path("summary")
public String summary() {
    models.put("orders", orderService.recentOrders());
    return "orders/summary.jsp";
}
```

The JSP receives `Models` entries as request attributes. It should render those values using JSP, Expression Language, and tag libraries; it should not open a database connection or repeat the controller's access-control decision.

Changing the View Folder Convention

An application can set `ViewEngine.VIEW_FOLDER` when its existing archive uses a different protected template root. The property affects every relative view name, so choose one convention and use it consistently.

```
@Override
public Map<String, Object> getProperties() {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    return Map.of(ViewEngine.VIEW_FOLDER, "/WEB-INF/pages/");
}
```

With that property, `return "orders/summary.jsp"` identifies `WEB-INF/pages/orders/summary.jsp`. Keep the root below `WEB-INF` unless direct access to templates is an intentional, separately protected design. If the application omits extensions through the standard view-extension property, document the convention and test every controller route after the change.

Providing a Custom View Engine

Provide a custom engine only when the application has a view type that JSP cannot serve. A custom engine is a CDI bean that implements `ViewEngine`. `supports` declares which view names it owns; `processView` writes the response for a selected name. The engine should have a clear suffix or other unambiguous selection rule so it cannot accidentally claim JSP views.

```
@ApplicationScoped
@Priority(ViewEngine.PRIORITY_APPLICATION)
public class TextViewEngine implements ViewEngine {
    @Override
    public boolean supports(String view) {
        return view.endsWith(".txt");
    }

    @Override
    public void processView(ViewEngineContext context)
            throws ViewEngineException {
        try {
            context.getOutputStream().write(
                String.valueOf(context.getModels().get("message"))
                    .getBytes(StandardCharsets.UTF_8));
        } catch (IOException exception) {
            throw new ViewEngineException(exception);
        }
    }
}
```

MVC orders built-in, framework, and application view engines using the defined priorities. An application engine normally uses `ViewEngine.PRIORITY_APPLICATION`. If two engines support the same name at the same priority, selection is not a useful application contract. Fix the suffixes or priorities rather than relying on incidental order.

Package the custom engine and its application-owned dependencies in the WAR. It can use normal CDI injection, but it must be safe for concurrent requests when it is application scoped and must convert failures into a meaningful view exception.

Diagnosing a View Failure

Start at the controller and move outward: confirm the selected action and returned view name; confirm the configured view folder; inspect the deployed WAR for the expected path; then inspect JSP compilation or the custom engine's failure. For a custom engine, also confirm CDI discovery, `supports` behavior, and non-conflicting priority. Preserve the original exception in logs rather than replacing it with a generic error view that hides the cause.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Internationalizing Jakarta MVC Applications

Learn how Jakarta MVC selects a request locale and how to make controller, form, and JSP behavior consistent for users in more than one locale.
This topic includes the following sections:

- Understanding the Request Locale
- Using the Default Locale Resolution
- Supplying an Application Locale Policy
- Localizing Views and Forms
- Testing Locale Behavior

Understanding the Request Locale

An MVC request has one resolved locale. `MvcContext` exposes it to a controller, and a JSP can use the MVC context while it renders the resulting view. Treat that locale as the presentation locale for the request: use it for message selection and display formatting, and keep it aligned with any human-entered values that MVC converts.

```
@Inject MvcContext mvc;

@GET
public String show() {
    models.put("locale", mvc.getLocale());
    return "locale.jsp";
}
```

This does not make every business value locale-specific. Persist money, quantities, and dates in their appropriate typed forms; format them for the view, and parse user input at the controller boundary.

Using the Default Locale Resolution

The default MVC locale resolver uses the request's `Accept-Language` header and falls back to the JVM default locale when it cannot select an applicable locale. This is a good starting point for a browser-facing application, but a production multilingual application should decide whether the JVM fallback is acceptable for every managed server.

Use the same expected locale in a controller and JSP; do not independently interpret headers in both places. That avoids one locale formatting a price while another locale selects the validation message.

Supplying an Application Locale Policy

Provide a CDI `LocaleResolver` when the application's policy is based on a signed-in user's preference, a trusted tenant configuration, or another application source. Give it an explicit priority above the default resolver when it should answer first, and return `null` when it has no applicable answer so the normal fallback can continue.

```
@ApplicationScoped
@Priority(100)
public class AccountLocaleResolver implements LocaleResolver {
    @Override
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    @Inject
     private AccountLocale accountLocale;
     public Locale resolveLocale(LocaleResolverContext context) {
        String preferred = accountLocale.findForCurrentUser();
        return switch (preferred) {
            case "fr" -> Locale.FRENCH;
            case "en" -> Locale.ENGLISH;
            default -> null;
        };
    }
}
```

Validate any request-selected language against an application allowlist. A locale resolver establishes presentation behavior for the whole request, so it should be fast, deterministic, and independent of an untrusted arbitrary locale value.

Localizing Views and Forms

Use the application's resource-bundle mechanism for labels and validation messages. Format a display value with the request locale immediately before rendering it, rather than passing a locale-formatted string to business logic.

```
NumberFormat formatter = NumberFormat.getCurrencyInstance(mvc.getLocale());
models.put("displayPrice", formatter.format(product.getPrice()));
```

For a form, test the exact input syntax that users see. A comma decimal separator, month order, or translated validation message may be correct in one locale and wrong in another. Keep the form field's type and constraint stable; only its human presentation and parsing are locale-sensitive.

Testing Locale Behavior

Test the default resolver with representative `Accept-Language` values, including quality values and a missing header. Test a custom resolver's priority and `null` fallback. For each supported locale, test at least one rendered message, one formatted value, and one submitted value that requires conversion. In a clustered deployment, verify that the chosen locale remains consistent after a new request reaches another managed server; persist a user preference when request headers alone cannot express the policy.

Using Form Method Override in Jakarta MVC Applications

Learn how an HTML form can invoke PUT, PATCH, or DELETE controller actions while retaining normal form security and validation behavior.
This topic includes the following sections:

- Understanding the HTML Form Limitation
- Creating an Overridden Action
- Configuring the Override Field
- Securing an Overridden Request
- Testing the Request Path

Copyright © 2007, 2026, Oracle and/or its affiliates.

Understanding the HTML Form Limitation

HTML forms directly submit GET or POST. Jakarta MVC can overwrite the method of a POST request when the request contains the configured hidden field. The default behavior is enabled and uses `_method`. This lets an application use a route such as `@PUT @Path("{id}")` without creating a second POST-only route solely for the browser.

Method override changes routing, not authorization. The browser still sends a POST, and the application must still protect the resulting state-changing action with normal Web application authorization, CSRF validation, binding, and business checks.

Creating an Overridden Action

The edit page posts a hidden method value and CSRF token. MVC processes the form according to the configured override behavior before selecting the controller action.

```
<form method="post" action="${mvc.uri('product-update', {'id': product.id})}">
  <input type="hidden" name="_method" value="PUT" />
  <input type="hidden" name="${mvc.csrf.name}" value="${mvc.csrf.token}" />
  <label for="name">Name</label>
  <input id="name" name="name" value="<c:out value='${product.name}'/>" />
  <button type="submit">Update</button>
</form>
```

```
@Inject
private BindingResult bindingResult;
@PUT
@Path("{id}")
@UriRef("product-update")
public String update(@PathParam("id") long id,
        @BeanParam @Valid ProductForm form) {
    if (bindingResult.isFailed()) {
        return "products/edit.jsp";
    }
    productService.update(id, form);
    return "redirect:products/" + id;
}
```

Only a POST form with the configured field participates in this feature. Do not assume that a GET request or an arbitrary request header has the same effect. When a JavaScript client can send a real PUT or PATCH request, prefer the normal HTTP method unless HTML-form compatibility is the reason for the override.

Configuring the Override Field

Set the portable form-method-overwriter properties on the application class when an application needs to disable the feature or use a different field name.

```
@Override
public Map<String, Object> getProperties() {
    return Map.of(
        FormMethodOverwriter.FORM_METHOD_OVERWRITE,
        FormMethodOverwriter.Options.ENABLED,
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
        FormMethodOverwriter.HIDDEN_FIELD_NAME, "_httpMethod");
}
```

Changing the name is a client contract change: update every JSP, automated test, and any non-browser client. Disable method override when the application does not use it rather than leaving a request transformation enabled without a purpose.

Securing an Overridden Request

Include the same CSRF token and validation logic that a direct POST action would use. A hidden field is client-controlled input and cannot authorize an operation. Test a valid overridden form, a missing override field, a non-POST request, an invalid method value, an invalid CSRF token, and an unauthorized caller. Assert both the matched route and the outcome: an update must not be performed when any of those preconditions fails.

Testing the Request Path

Test method override through the deployed application rather than by calling the controller method directly. Verify all of the following cases:

- A POST request containing the configured override field and `PUT` value reaches the `@PUT` controller action.
- A POST request without the override field remains a POST request.
- A non-POST request is not overwritten.
- An unsupported method value is rejected and does not update application data.
- A missing or invalid CSRF token is rejected.
- An unauthorized caller cannot perform the update.

For every rejected request, verify that the service method that changes application data is not called.

Jakarta MVC Implementation Reference for Oracle WebLogic Server

Use this reference when moving from a Jakarta MVC 3.0 specification topic to the corresponding Web application concern on Oracle WebLogic Server 26.1.
Review the following sections.

Implementation Boundary

Oracle WebLogic Server 26 supplies Jakarta MVC 3.0 with Eclipse Krazo as the default MVC implementation. Applications compile against the Jakarta EE and Jakarta MVC APIs and use the standard controller, CDI, provider, view-engine, and configuration contracts. The implementation name explains the server runtime supplied for MVC; it is not an application dependency or a proprietary controller API.

Package application code, templates, descriptors, and application-owned libraries in the WAR. Keep the MVC API at provided scope in the application build and avoid using the archive to select a different MVC implementation.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Specification-to-Application Map

Jakarta MVC 3.0 area	Application task on Oracle WebLogic Server	Reference Link
Controllers and controller instances	Use CDI-managed REST resource methods with `@Controller`; select a view, response, or redirect.	Using Models, Views, and Controllers
Models, views, URI building, redirects	Put request display data in `Models` or named beans; render protected JSPs; use `@UriRef` for stable links.	Using Models, Views, and Controllers
Binding and conversion	Use a `@BeanParam` form with `@MvcBinding`, Bean Validation, and `BindingResult`.	Using Data Binding in Jakarta MVC Applications
Security	Configure Web application authorization and use MVC CSRF and output encoding.	Securing Jakarta MVC Applications
Events	Observe lifecycle facts with short, synchronous CDI observers.	Using Events in Jakarta MVC Applications
Applications, context, properties, providers	Define the application path, use `MvcContext`, configure portable properties, and add application providers with `@Provider`.	Configuring Jakarta MVC Applications
View engines	Use JSP by default or supply an unambiguous CDI `ViewEngine`.	Using View Engines in Jakarta MVC Applications
Internationalization	Use the MVC request locale and, where needed, a CDI `LocaleResolver`.	Internationalizing Jakarta MVC Applications
Form method override	Use a POST form plus the configured hidden field for a PUT, PATCH, or DELETE action.	Using Form Method Override in Jakarta MVC Applications

Common Annotations

Annotation	Purpose
@Controller	Identifies a Jakarta RESTful Web Services resource method or class as MVC.
@View	Selects the view for a `void` action or the default when a non-void action returns `null`.
@MvcBinding	Lets an action receive binding and validation failures through `BindingResult`.
@CsrfProtected	Marks an action for CSRF validation in explicit CSRF mode.
@RedirectScoped	Makes a CDI bean available across a redirect and the immediately following request.
@UriRef	Assigns a stable logical route reference for URI generation.
@Provider	Makes an application JAX-RS provider eligible for provider discovery.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Standard Properties

Set these properties through the Jakarta RESTful Web Services application configuration mechanism. Treat a changed security, view-folder, or client-header property as an application behavior change and cover it with a deployed integration test.

Constant	Default or Use
ViewEngine.VIEW_FOLDER	The default JSP view folder is /WEB-INF/views/; set the property for another protected root.
ViewEngine.VIEW_EXTENSION	Establishes a default view extension when the application uses extension-less names.
Csrf.CSRF_PROTECTION	Jakarta MVC 3.0 defaults to implicit CSRF protection.
Csrf.CSRF_HEADER_NAME	Names the request header used when a client submits a CSRF token by header.
FormMethodOverwriter.FORM_METHOD_OVERWRITE	Form method override is enabled by default.
FormMethodOverwriter.HIDDEN_FIELD_NAME	The default hidden field is `_method`.

Archive Checklist

Before delivery, verify one end-to-end route for each feature used by the application: matching and JSP rendering, valid and invalid form submission, CSRF behavior, authorization, redirects, generated URI, locale, any custom view engine, and any provider that affects a response. This provides evidence for the deployed application rather than assuming that a unit test of a controller covers the Web container and MVC integration.

```
WEB-INF/
├── beans.xml                  CDI discovery configuration when needed
├── classes/                   application controllers, forms, and
services
├── lib/                       application-owned libraries
├── views/                     JSP views, normally below this protected
root
├── web.xml                    optional standard Web configuration
└── weblogic.xml               optional WebLogic Server deployment
settings
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

4

Creating and Configuring Servlets

Learn about what is new and changed in recent servlet specifications, and how to create and configure servlets.
Review the following sections.

What's New and Changed in Servlets in 26.1.0.0.0

These sections summarize the changes in the servlet programming model and requirements.

What Was New and Changed in Servlet 6.1

WebLogic Server supports the Jakarta Servlet 6.1 specification (see https://jakarta.ee/specifications/servlet/6.1/), which introduces the following new features:

- Allows control of the status code and response body when sending a redirect.
- Adds a query-string attribute to error dispatches.
- Adds constants for new HTTP status codes.
- Adds overloaded methods that accept `Charset` instead of a character-set name as a `String`.
- Adds `ByteBuffer` support to `ServletInputStream` and `ServletOutputStream`.
- Removes references to `SecurityManager` and its associated APIs.

Configuring Servlets

Learn how to configure servlets using Jakarta EE metadata annotations versus deployment descriptors, and how to use servlet mapping in a Web application.

Servlet Annotations

With Jakarta EE metadata annotations, the standard `web.xml` deployment descriptor is optional. The servlet specification states annotations can be defined on certain Web components, such as servlets, filters, listeners, and tag handlers. The annotations are used to declare dependencies on external resources. The container will detect annotations on such components and inject necessary dependencies before the component's life cycle methods are invoked. See WebLogic Annotation for Web Components.

However, you can also define servlets as a part of a Web application in several entries in the 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Table 4-1) showing the URLs used to invoke these servlets.

Example 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`

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 4-1 (Cont.) url-patterns and Servlet Invocation

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

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.
2. Add a servlet-mapping with url-pattern = "/" as follows:

```
<servlet-mapping>
<servlet-name>MyOwnDefaultServlet</servlet-name>
<url-pattern>/</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`.
   c. Using the `<servlet-mapping>` element, map file extensions to the `myFileServlet` (in addition to the mappings for your default servlet). For example, to use `myFileServlet` to serve GIF files, map the `*.gif` URL pattern to `myFileServlet`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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:

Example 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

Examine 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 Jakarta EE 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 jakarta.servlet.*;
   import jakarta.servlet.http.*;
   import java.io.*;
   ```

2. Extend `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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:

   - The name of the Web application containing the servlet and
   - 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` represents the deployed context root of the Web application.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- `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

Examine the 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
- Application design—HTTP servlets offer many ways to design your application. The following sections provide detailed information about writing servlets:
  - Providing an HTTP Response
  - Threading Issues in HTTP Servlets
  - Dispatching Requests to Another Resource
- 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
- 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.* See the following sections:
  - Session Tracking from a Servlet
  - Using Cookies in a Servlet
  - Configuring Session Persistence
- 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 JDBC drivers, JDBC database connection pools, Jakarta Messaging (JMS), Enterprise JavaBeans (EJB), and Remote Method Invocation (RMI). See the following sections:
  - Using WebLogic Services from an HTTP Servlet
  - Accessing Databases

Complete HelloWorldServlet Example

Examine 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Example 4-3 HelloWorldServlet.java

```
import jakarta.servlet.*;
import jakarta.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;charset=UTF-8");
    // 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 `ORACLE_HOME\wlserver\samples\server\examples\src\examples\splitdir\helloWorldEar` directory of your WebLogic Server distribution, where`ORACLE_HOME` represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Debugging Servlet Containers

Learn about the debugging options available in the WebLogic Server servlet container:

Disabling Access Logging

Logging access for servlets can be expensive with regard to server performance. Therefore, in cases where access logging is not required, you can improve performance by disabling logging to the access log file.

Usage

The optional `access-logging-disabled` property in the `container-descriptor` in `weblogic.xml` can be used to specify whether access logging for an underlying Web application is disabled.

- If the property is set as `true`, then application accesses are *not* logged.
- If the property is not defined or is set as `false`, then application accesses are logged.

> **Note**
>
> The `access-logging-disabled` property functions at the Web application level. Therefore, if it is defined in a Web application, it does not affect other Web applications. This property works under both development mode and production mode.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> To disable logging for internal applications, use this property, `weblogic.servlet.logging.LogInternalAppAccess=false`.

Example

The following example demonstrates how to disable access logging:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app">
<container-descriptor>
<access-logging-disabled>true</access-logging-disabled>
</container-descriptor>
</weblogic-web-app>
```

Debugging Specific Sessions

Tracking session change is very helpful when developing applications, especially for replicated sessions. Although you can utilize `HttpSessionAttributeListener` to track session changes at the Web application level, developers need a finer-grained debugging option to track session changes during a specific request.

Usage

The `wl_debug_session` request attribute or a same-named session attribute can log attribute changes in the current session. When either flag is used, the container logs the modifications of the underlying session in the server log.

You can enable specific session debugging by using either of the following methods:

- Set the `wl_debug_session` attribute to the current session, as follows:
- `session.setAttribute('wl_debug_session', Boolean.TRUE);`
- Use the `wl_debug_session` attribute in the request query string as the indicator. The container adds a `wl_debug_session` session attribute to the current session, as shown in the following example:

  ```
  http://localhost/foocontext/foo?wl_debug_session
  ```

To stop debugging a session, you can simply remove the `wl_debug_session` attribute.

> **Note**
>
> This feature is available only in development mode. The severity of the debug message is at the `debug` level. You need to adjust the severity of the logger to `debug` or lower for the system logger to output the debug message to the server log file.

Tracking a Request Handle Footprint

Tracking a request handle footprint is very helpful while in application development mode. For example, when debugging an application, you need to know many pieces of information. This includes such information as: what request is received, how it is dispatched, what session it is

Copyright © 2007, 2026, Oracle and/or its affiliates.

bound to it, when the servlet is invoked, and what response is sent. Finally, when a `ServletException` occurs, you need a way to link the exception to corresponding request to find the root cause of the error.

Usage

The WebLogic Server servlet container provides more detailed log messages during request handling to better describe each milestone in a request flow. No additional configuration changes are required other than enabling the `DebugHttp` logger.

You can then find the footprint of a request handle in the server log. Once in production mode, you should disable `DebugHttp` logger to maximize server performance.

Copyright © 2007, 2026, Oracle and/or its affiliates.

5

Creating and Configuring JSPs

Learn how to create and configure Jakarta Server Pages (JSPs).
This chapter includes the following sections:

For HTTP error-response configuration, request character encoding, and IANA-to-Java character-set mapping, see Customizing HTTP Error Responses, Determining the Encoding of an HTTP Request, and Mapping IANA Character Sets to Java Character Sets in Chapter 2.

Configuring Jakarta Server Pages (JSPs)

In order to deploy JSPs 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.

Registering a JSP as a Servlet

You can register a JSP as a servlet using the `servlet` element of the Jakarta EE 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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>
<taglig-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 *Developing JSP Tag Extensions for Oracle WebLogic Server*.

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. See:

- Using Custom WebLogic JSP Tags (cache, process, repeat)
- 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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:

Example 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 welcome files, see the servlet 6.1 specification, section 10.10 at https://jakarta.ee/specifications/servlet/6.1/jakarta-servlet-spec-6.1#welcome-files.

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 Jakarta EE 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 form, declare the character encoding accepted by the server with the HTML `accept-charset` attribute. Use UTF-8 unless the application has a documented legacy encoding requirement. For example:

  ```
  <form method="post"
        action="https://some.example.com/myWebApp/foo/index.html"
        accept-charset="UTF-8">
    ...
  </form>
  ```

  The application must use the same encoding when it reads the request. When a WebLogic Server-specific path-to-encoding mapping is required, configure the `input-charset` element in `weblogic.xml`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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`.

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

Mapping IANA Character Sets to Java Character Sets

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

To map an IANA character-set name to a Java character-set name, 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.

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:

Example 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 12, Mapping Requests to Servlets of the servlet 6.1 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.

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 3.0 specification at `https://jakarta.ee/specifications/pages/3.0/jakarta-server-pages-spec-3.0#jsp-configuration`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Writing JSP Documents Using XML Syntax

A JSP document is a JSP page written in XML syntax. JSP documents typically use the .jspx extension, and tag files written in XML syntax use .tagx. The container checks JSP documents for well-formed XML while applying normal JSP processing semantics. A JSP document is different from a Jakarta Faces Facelets view, which normally uses the .xhtml extension.

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`.

Example 5-3 Simple JSP Document

```
<table>
<c:forEach
xmlns:c=xmlns:c="jakarta.tags.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.
- 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

6

Using JSF and JSTL

Learn how to use Jakarta Server Faces (JSF) and Jakarta Standard Tag Library (JSTL) with WebLogic Server.
This chapter includes the following sections:

Using JSF and JSTL With Web Applications

Jakarta Server Faces (JSF) and Jakarta Standard Tag Library (JSTL) are an integral part of Jakarta EE and, as such, are incorporated directly into WebLogic Server. All Jakarta EE technologies are present on the WebLogic Server classpath. No additional configuration is required to use any of the Jakarta EE technologies in your applications. Applications deployed to WebLogic Server can seamlessly make use of JSF 4.1 and JSTL 3.0 without requiring you to deploy and reference separate shared libraries.

The Jakarta EE API JAR file is included in `WL_HOME\wlserver\server\lib\jakarta.javaee-api-11.1.0.jar`, where `WL_HOME` represents the top-level installation directory for WebLogic Server.

For information about referencing these shared libraries with your Web applications, see Creating Shared Jakarta EE Libraries and Optional Packages in *Developing Applications for Oracle WebLogic Server*.

Jakarta Server Faces (JSF)

JSF technology simplifies building user interfaces for JavaServer applications. Developers of various skill levels can quickly build Web applications by: assembling reusable UI components in a page, connecting these components to an application data source, and wiring client-generated events to server-side event handlers.

Facelets Views

Jakarta Faces applications normally define views with Facelets and use the `.xhtml` file extension. Facelets is the current view-definition technology for Jakarta Faces. It is distinct from JSP pages (`.jsp`) and JSP documents (`.jspx`). The Jakarta MVC chapter uses JSP because Jakarta MVC 3.0 requires a built-in JSP view engine; that requirement does not make Facelets the default view engine for Jakarta MVC.

WebLogic Server supports the JSF 4.1 specification at https://jakarta.ee/specifications/faces/4.1/.

WebLogic Server's Jakarta Faces implementation supports capabilities including:

- CDI integration, including the CDI-compatible `@ManagedProperty`.
- WebSocket integration through `<f:websocket>`.
- Ajax method invocation.
- Class-level Bean Validation through `<f:validateWholeBean>`.
- Java date and time support.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- Component search expressions.

If you selected to install the server examples with your WebLogic Server installation, the sample application demonstrates the following key features of JSF:

- Ajax method invocation using the new `<h:commandScript>` component
- CDI converter
- Iterable `h:dataTable`
- CDI for evaluation of JSF-specific EL implicit Objects
- CDI injection
- New Java date and time
- `@ManagedProperty`
- Iterable `ui:repeat`
- Class-level bean validation using `f:validateWholeBean`
- CDI validator
- WebSocket using `f:websocket`

The JSF examples are located in the *`ORACLE_HOME`*`\wlserver\samples\server\examples\src\examples\javaee8\jsf` directory, where *`ORACLE_HOME`* represents the directory in which you installed WebLogic Server.

For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Jakarta Standard Tag Library (JSTL)

The Jakarta Standard Tag Library (JSTL) encapsulates as simple tags the core functionality common to many Web applications. JSTL has support for common, structural tasks, such as:

- Iteration and conditionals
- Tags for manipulating XML documents
- Internationalization tags
- SQL tags

JSTL also provides a framework for integrating existing custom tags with JSTL tags.

WebLogic Server supports the JSTL 3.0 specification at https://jakarta.ee/specifications/tags/3.0/.

JSF Backward Compatibility

A Faces 2.x application can require migration before deployment on the Jakarta EE 11 runtime. In particular, application classes and libraries that use `javax.*` APIs must be migrated to the corresponding `jakarta.*` APIs unless the application uses an explicitly supported WebLogic Server compatibility mechanism. Remove any application-bundled Faces implementation before testing against the server-provided Faces 4.1 implementation.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> The `jsf-4.1.war` deployable library, included in WebLogic Server, is empty, as applications built for JSF 2.0 will continue to run unmodified using the built-in JSF 4.1 implementation of WebLogic Server 26.1.0.0.0.
>
> WebLogic Server includes the empty `jsf-4.1.war` library to avoid any software that depends on its existence. You can leave references to the library unchanged without harm. However, Oracle recommends removing any references to this empty library, as these references add no functionality.
>
> See the following sections:

Deploying JSF and JSTL Libraries

> **Note**
>
> In this release of WebLogic Server, you can deploy JSF 2.3 and JSTL 1.2 applications directly. For backward compatibility, use the following directions when deploying JSF 1.x and JSTL 1.1 applications.

When deploying JSF 1.2 applications, use the JSF and JSTL libraries which are provided as Web application libraries. You must deploy the libraries before deploying the Web application that is using JSF 1.2 or JSTL functionality. You can deploy the libraries using the WebLogic Remote Console or the command-line `weblogic.Deployer` utility.

Here's an example of deploying a JSF 1.2 library using the `weblogic.Deployer` command-line:

> **Note**
>
> Do not place an administrative password directly on the command line. Use the supported WebLogic Server user configuration and key-file mechanism, or enter credentials through the approved deployment workflow.

```
java weblogic.Deployer -adminurl t3://localhost:7001
  -deploy -library
 d:/oracle_home/wlserver/common/deployable-libraries/jsf-1.2.war
```

This command deploys the JSF 1.2 library using the default `library-name`, `specification-version` and `implementation-version` defined by the `MANIFEST.MF` in the library.

After a library is deployed, the `extension-name`, `specification-version` and `implementation-version` of the library can be found in the Remote Console. This information can also be found in the `MANIFEST.MF` file of the library `WAR` file.

For more information about deploying a Web module, see Preparing Applications and Modules for Deployment in *Deploying Applications to Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Referencing a JSF or JSTL Library

To reference a JSF or JSTL library, a standard Web application can define a `<library-ref>` descriptor in the application's `weblogic.xml` file. Here is an example:

```
<library-ref>
  <library-name>jsf</library-name>
  <specification-version>1.2</specification-version>
  <implementation-version>1.2</implementation-version>
  <exact-match>false</exact-match>
</library-ref>
```

For more information on referencing a Web application library, see Creating Shared Jakarta EE Libraries and Optional Packages in *Developing Applications for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

7

Configuring Resources in a Web Application

Learn how to configure Web application resources in WebLogic Server.
This chapter includes the following sections:

Configuring Resources in a Web Application

The resources that you use in a Web application are generally deployed externally to the Web application. JDBC data sources 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 for more information.

You can also deploy JDBC data sources 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 their 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>` elements as explained in Configuring Resources.

For a WAR that uses a NoSQL-backed Jakarta Data repository, see Referencing a NoSQL System Resource. Package the reference declaration and JNDI mapping in the WAR; configure the global NoSQL system resource separately in the domain. For repository programming guidance, see *Developing Jakarta Data Applications for Oracle WebLogic Server*.

Configuring Resources

When accessing resources such as a data source 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 *Administering JMS Resources for Oracle WebLogic Server*.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

The following example illustrates how to use an external data source. It assumes that you have defined a data source called `accountDataSource`. See Create a Generic Data Source in the *Oracle WebLogic Remote Console Online Help*.

Example 7-1 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.

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

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 Jakarta EE 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

With the addition of the `ejb-link` syntax, the `ejb-reference-description` 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-description` 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 run time through a JNDI name lookup relative to `java:comp/env`.

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

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.)

Example 7-2 Example Deployment Descriptor

```
<application>
   <display-name>MyApp</display-name>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    <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 Jakarta EE 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:

Example 7-3 <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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

Example 7-4 <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 run time, 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.

> **Note**
>
> **Compatibility guidance:** `ClasspathServlet` has been retained for existing applications. For new applications, use the standard asynchronous processing model in Jakarta Servlet 6.1.

> **Note**
>
> **Security Warning:** `ClasspathServlet` can expose resources available to the application class loader. Map it only to a reviewed, application-owned resource location. Do not use a broad mapping that can expose configuration, classes, libraries, or server resources. For new applications, package static content in the Web application and serve it through the normal Web resource path.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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
```

> **Note**
>
> WebLogic Server provides a secured production mode that enforces more restrictive and stringent security settings to ensure less vulnerability to threats. The `ServerTemplateMBean` includes a `ClasspathServletSecureModeEnabled` attribute that, when secure mode is enabled, will serve only class files from well known packages required for JDBC and JMS functionality.
>
> If secure mode is disabled, do not place any resources or classes that should not be publicly available in any of the locations listed above that the `ClasspathServlet` serves.

Using CGI with WebLogic Server

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.

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

See Configuring How a Client Accesses a Web Application.

> **Note**
>
> WebLogic Server provides functionality to support your legacy Common Gateway Interface (CGI) scripts. For new projects, Oracle recommends that you use HTTP servlets or JSPs.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta EE 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.

Example 7-5 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>
   /wlserver26.1/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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Where

host:port—Host name and port number of WebLogic Server.

myWebApp—Name of your Web application.

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

myscript.pl—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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

8

WebLogic Annotation for Web Components

Learn how to annotate Web components in WebLogic Server.
This chapter includes the following sections:

Servlet Annotation and Dependency Injection

The Servlet 6.1 specification provides annotations to enable declarative-style programming.

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

> **Note**
>
> WebLogic Server-specific annotations have been deprecated and will be removed in a future release: `@WLServlet`, `@WLFilter`, and `@WLInitParam`, in favor of the standard annotations defined in the Servlet 6.1 specification. Also, instead of `weblogic.servlet.http.AbstractAsyncServlet`, you should use the standard asynchronous processing model defined in the Servlet 6.1 specification.

The servlet specification states that annotations can be defined on certain Web components, such as servlets, filters, listeners, and tag handlers. The annotations are used to declare dependencies on external resources. The container will detect annotations on such components and inject necessary dependencies before the component's life cycle methods are invoked. Dependency Injection (DI) will only be done on certain components, as described in Web Component Classes That Support Annotations.

Annotation processing and DI will be performed on all Web applications that have the version set to 2.5 or higher. However, annotation processing is expensive and it can increase the deployment time for Web applications depending on the size of the included classes. Set the `metadata-complete` attribute to `true` in the `web.xml` descriptor if your Web application does not have any annotations *and* if you have the version set to 2.5 or higher to avoid unnecessary scanning of the Web applications classes for annotations. Alternatively, you can turn off annotation processing and DI for all the Web applications by setting `-Dweblogic.servlet.DIDisabled=true` flag when starting WebLogic Server.

For more information about using Jakarta EE annotations and dependency injection with WebLogic Server applications, see Using Jakarta EE Annotations and Dependency Injection and Using Contexts and Dependency Injection for the Jakarta EE Platform in *Developing Applications for Oracle WebLogic Server*. For detailed information about EJB-specific annotations for WebLogic Server EJBs, see *Developing EJBs for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

If you selected to install the server examples, you will find this annotation code example, "Using Annotations for Servlets, Filters and Listeners," in the *ORACLE_HOME*\wlserver\samples\server\examples\examples\src\examples\javaee11\servlet\annotation directory of your WebLogic Server distribution, where *ORACLE_HOME* represents the directory in which you installed the WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Web Component Classes That Support Annotations

This section describes the behavior of annotations and dependency injection (DI) of resources in a Jakarta EE compliant Web container.

The Web container only processes annotations for the types of classes listed in Table 8-1.

Table 8-1 Web Components and Interfaces Supporting Annotations and Dependency Injection

Component Type	Interfaces
Servlets	`jakarta.servlet.Servlet`
Filters	`jakarta.servlet.Filter`

Listeners	`jakarta.servlet.ServletContextListener`
	`jakarta.servlet.ServletContextAttributeListener`
	`jakarta.servlet.ServletRequestListener`
	`jakarta.servlet.ServletRequestAttributeListener`
	`jakarta.servlet.http.HttpSessionListener`
	`jakarta.servlet.http.HttpSessionAttributeListener`
	`jakarta.servlet.AsyncListener`

Tag handlers	`jakarta.servlet.jsp.tagext.SimpleTag` `jakarta.servlet.jsp.tagext.BodyTag`

The Web container will not process annotations on classes like Jakarta Enterprise Beans and other helper classes. The Web container follows these steps to achieve DI:

1. Annotation Processing—The Web container processes annotations during the Web application deployment phase. As annotations are processed, the container figures out the relevant entries in the descriptor that get affected by the annotation and updates the descriptor tree. The servlet specification indicates that all annotations can be declared in the descriptor by defining an injection target. The Web container updates the descriptor tree with the injection targets so that as deployment continues the JNDI tree is updated with the necessary entries.
2. Dependency Injection (DI)—DI is done when instances are created (for the types listed in Table 8-1). For listeners and filters, this occurs during the deployment phase, and for servlets it can occur during deployment or run time.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> In any Web application component, if one DI fails, it will cause all subsequent DIs upon the same component to be ignored.

Annotations Supported By a Web Container

Table 8-2 lists all the annotations that must be supported by the Web container.

Table 8-2 List of Supported Annotations

@Annotation
DeclaresRoles
EJB
EJBs
PersistenceContext
PersistenceUnit
PersistenceUnits
PersistenceContexts
PostConstruct
PreDestroy
Resource
Resources
WebServiceRef
WebServiceRefs
RunAs

The Web container makes use of the Jakarta EE container's annotation processing and dependency injection mechanisms to achieve this functionality.

The specification states that the Web container should not process annotations when `metadata-complete` attributes are set to `true` in the `web.xml` descriptor. If annotations are properly defined and annotation processing succeeds and dependencies are properly injected, the annotated fields are initialized properly and annotated methods are invoked at the proper phase in the life cycle. If DI fails, these annotated fields will be `null`.

> **Note**
>
> If multiple methods in a Web component class, such as a servlet, filter, and such, are annotated with `PostConstruct` or `PreDestroy`, then the Web component will fail to deploy such an application. Similarly, if an EJB component class, such as a session bean, is annotated with `PostConstruct` or `PreDestroy`, or an EJB interceptor is annotated with `PostConstruct`, `PreDestroy`, `PostActivate`, or `PrePassivate`, then the EJB component will also fail to deploy such an application.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Fault Detection and Recovery

Any failure during annotation processing will yield a deployment exception that will prevent deployment of the Web application. If a failure happens during DI, the container will log a warning message in the server logs indicating the reason for the failure. The annotated fields in the instance of the class will be `null` and any life cycle annotated methods will not be invoked in case of DI failure.

Limitations

The WebLogic servlet container supports annotations on Web components that are declared in the `web.xml` descriptor. Any listeners, filters or servlets registered dynamically via the `weblogic.servlet.WeblogicServletContext` method will not have their annotations processed and no DI will be done for such components.

Deprecated WebLogic Server-Specific Servlet Annotations

`@WLServlet`, `@WLFilter`, and `@WLInitParam` are deprecated and will be removed in a future release.

Do not use the deprecated annotations in new applications. Use `@WebServlet`, `@WebFilter`, `@WebInitParam`, and `@WebListener`.

```
import jakarta.servlet.annotation.WebInitParam;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;

@WebServlet(
    name = "ExampleServlet",
    urlPatterns = {"/example/*"},
    initParams = {
        @WebInitParam(name = "mode", value = "standard")
    }
)
public class ExampleServlet extends HttpServlet {
}
```

Use `@WebFilter` for filters and `@WebListener` for supported listener classes. Use `web.xml` when descriptor-based configuration or ordering is required.

WLServlet

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

Copyright © 2007, 2026, Oracle and/or its affiliates.

You can annotate a servlet class with `WLServlet` annotation (`weblogic.servlet.annotation.WLServlet`). This annotation defines various attributes for declaring parameters for the servlet. All attributes on this annotation are optional.

Attributes

Table 8-3 Attributes of WLServlet Annotation

Name	Description	Data Type	Required?
displayName	Display name for the servlet after deployment	String	No
description	Servlet description	String	No
icon	Icon location	String	No
name	Servlet name	String	No
initParams	Initialization parameters for the servlet	WLInitParam[]	No
loadOnStartup	Whether the servlet should load on startup	int	No
runAs	The run-as user for the servlet	String	No
mapping	The url-pattern for the servlet	String[]	No

Example 8-1 illustrates the usage of the annotation in a servlet class.

Example 8-1 WLServlet Annotation

```
@WLServlet (
   name = "FOO",
   runAs = "SuperUser"
   initParams = { @WLInitParam (name="one", value="1") }
   mapping = {"/foo/*"}
)
. . .
```

The WebLogic servlet container detects the annotation and installs this servlet for deployment. During the annotation processing phase of the Web applications deployment, the descriptor bean corresponding to `web.xml` descriptor is updated with the relevant entries corresponding to the annotation.

Example 8-2 shows how the descriptor bean looks after being updated.

Example 8-2 Updated web.xml Descriptor

```
<web-app>
. . .
   <servlet>
      <servlet-name>FOO</servlet-name>
      <servlet-class>my.TestServlet</servlet-class>
      <init-param>
         <param-name>one</param-name>
         <param-value>1</param-value>
      </init-param>
   </servlet>
   <servlet-mapping>
      <servlet-name>FOO</servlet-name>
      <url-pattern>/foo/*</url-pattern>
   </servlet-mapping>
. . .
</web-app>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

Fault Detection And Recovery

Any error during the processing of this annotation will result in a deployment error with a proper message in the server logs.

WLFilter

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

You can annotate a filter class with `WLFilter` annotation (`weblogic.servlet.annotation.WLFilter`). This annotation defines various attributes for declaring parameters for the filter. All attributes on this annotation are optional.

Attributes

Table 8-4 Attributes of WLFilter Annotation

Name	Description	Data Type	Required?
displayName	Display name for the filter after deployment	String	No
description	Filter description	String	No
icon	Icon location	String	No
name	Filter name	String	No
initParams	Initialization parameters for the filter	WLInitParam[]	No
mapping	The url-pattern for the filter	String[]	No

Example 8-3 illustrates the usage of the annotation in a filter class.

Example 8-3 WLFilter Annotation

```
@WLFilter (
   name = "BAR",
   initParams = { @WLInitParam (name="one", value="1") }
   Mapping = {"/bar/*"}
)
. . .
```

The WebLogic servlet container detects the annotation and installs this filter for deployment. During the annotation processing phase of the Web application deployment, the descriptor bean corresponding to `web.xml` descriptor is updated with the relevant entries corresponding to the annotation.

Example 8-4 shows how the descriptor bean looks after being updated.

Example 8-4 Updated web.xml Descriptor

```
<web-app>
. . .
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
   <filter>
      <filter-name>BAR</filter-name>
      <filter-class>my.TestFilter</filter-class>
      <init-param>
         <param-name>one</param-name>
         <param-value>1</param-value>
      </init-param>
   </filter>
   <filter-mapping>
      <filter-name>BAR</filter-name>
      <url-pattern>/bar/*</url-pattern>
   </filter-mapping>
. . .
</web-app>
```

Fault Detection and Recovery

Any error during the processing of this annotation will result in a deployment error with a proper message in the server logs.

WLInitParam

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

You can use the `@WLInitParam` annotation (`weblogic.servlet.annotation.WLInitParam`) to specify the initial parameters for servlets and filters declared using the `@WLServlet` and `@WLFilter` annotations.

Attributes

Table 8-5 Attributes of WLFilter Annotation

Name	Description	Data Type	Required?
name	The initial parameter name.	String	No
value	The initial parameter value.	String	No

Example 8-5 provides an example of `WLInitParam` annotation.

Example 8-5 Example WLInitParam Annotation

```
initParams = {@WLInitParam(name="one", value="1"),
              @WLInitParam(name="two", value="2")}
```

Annotating a servlet or filter class with the above annotation is equivalent to declaring the init params in Example 8-6 in the `web.xml` descriptor.

Example 8-6 Init Params In web.xml

```
. . .
<init-param>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  <param-name>one</param-name>
  <param-value>1</param-value>
</init-param>
<init-param>
  <param-name>two</param-name>
  <param-value>2</param-value>
</init-param>
. . .
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

9

Servlet Programming Tasks

Learn how to write HTTP servlets in a WebLogic Server environment.
This chapter includes the following sections:

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. See Servlet Best Practices.
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.)

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 Jakarta EE 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 Jakarta EE 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

To retrieve initialization parameters, call the `getInitParameter(String name)` method from the parent `jakarta.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 Jakarta EE 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

Learn 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");
     ```


Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 `jakarta.servlet.http.HttpServletRequest` object and one of the methods shown in the following two examples:

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

   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, Oracle 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.

      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.

   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 `jakarta.servlet.ServletResponse` interface. See the Servlet 6.1 specification at https://jakarta.ee/specifications/servlet/6.1/.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Do not assume that a request parameter is present. `getParameter(name)` returns one value or `null`. `getParameterValues(name)` returns all values as a `String[]` or returns `null` when the parameter is absent. Check for `null` before using the returned value or array.

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);
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  System.out.println("\nParameter name is " + paramName);
  if (paramValues != null) {
    for (int i = 0; i < paramValues.length; i++) {
      System.out.println(", value " + i + " is " +
                         paramValues[i]);
    }
  }
}
```

> **Note**
>
> Encode untrusted data for the output context before returning it. Using the wrong encoder can leave the application vulnerable to cross-site scripting. For more information, refer to Securing Client Input in Servlets.

Methods for Using the HTTP Request

This section defines the methods of the `jakarta.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 `jakarta.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 Example 9-1, the `HelloWorld2.java` servlet example is modified to accept a user name as a query parameter, in order to display a more personal greeting. The `service()` method is shown here.

Example 9-1 Retrieving Input with the service() Method

```
public void service(HttpServletRequest req,
        HttpServletResponse res)
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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.

See *Cross Site Scripting Prevention Cheat Sheet* on the Open Web Application Security Project (OWASP) website at https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Encode untrusted data for the context in which it is inserted. Do not rely on a manually maintained list of characters. HTML text, HTML attributes, URLs, JavaScript, and CSS are different output contexts and can require different encoders.

Compatibility: Using a WebLogic Server Utility Method

> **Note**
>
> **Compatibility guidance:** The `weblogic.servlet.security.Utils.encodeXSS()` method is retained for existing code. It does not replace context-specific encoding.

For existing code that inserts untrusted data into HTML text, WebLogic Server provides `weblogic.servlet.security.Utils.encodeXSS()`. Review each output location separately because HTML attributes, URLs, JavaScript, and CSS can require different encoders. For example:

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

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 `jakarta.servlet.http.Cookie` object.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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>example.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 `example.com`. For more information about this property or configuring session cookies, see Setting Up Session Management .

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 certain 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 Jakarta EE 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`—The name of the cache. It defaults to the request URI for compatibility with `*.extension` URL patterns.
- `Timeout`—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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

The scope defaults to *application*. To use *cluster* scope you must set up the *ClusterListener*.

- `Key`—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`—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`—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:

- *Developing Jakarta Enterprise Beans for Oracle WebLogic Server*
- *Developing JDBC Applications for Oracle WebLogic Server*
- *Developing JNDI Applications for Oracle WebLogic Server*
- *Developing JMS Applications for Oracle WebLogic Server*

Accessing Databases

WebLogic Server supports the use of 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 *Developing JDBC Applications for Oracle WebLogic Server*.

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

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 Remote Console.

> **Note**
>
> Using a `DataSource` object is recommended when creating Jakarta EE-compliant applications.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using a Data Source in a Servlet

1. Register a connection pool using the Remote Console, **Services > Data Sources > 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

Direct driver connections are included for specialized or compatibility cases. For WebLogic Server applications, use a configured `DataSource` so that the server can provide connection pooling, transaction integration, and lifecycle management.

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. Oracle provides JDBC drivers for Oracle and Microsoft SQL Server. See *Developing JDBC Applications for Oracle WebLogic Server*.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Dispatching Requests to Another Resource

Read 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 9.2 of the Servlet 6.1 specification (see https://jakarta.ee/specifications/servlet/6.1/jakarta-servlet-spec-6.1#using-a-request-dispatcher).

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);`
   - where *`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 Jakarta EE 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);` See Forwarding a Request.
   - `rd.include(request,response);` See Including a Request.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Servlet 2.3 and older specifications did not specify whether filters should be applied on forwards and includes. As of Servlet 2.4, the 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 26.1, the default is `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 Filters and the RequestDispatcher, see the Servlet 6.1 specification at https://jakarta.ee/specifications/servlet/6.1/jakarta-servlet-spec-6.1#filters-and-the-requestdispatcher. For more information about writing and configuring filters for WebLogic Server, see Filters.

Proxying Requests to Another Web Server

Learn how to proxy HTTP requests to another Web server:

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Example 9-2). 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 *Command Reference for Oracle WebLogic Server*.
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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

5. Deploy the Web application on the WebLogic Server instance that redirects incoming requests.

Sample Deployment Descriptor for the Proxy Servlet

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

Example 9-2 Sample web.xml for Use with ProxyServlet

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

<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                             https://jakarta.ee/xml/ns/jakartaee/web-app_6_1.xsd"
         version="6.1">

<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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>

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

</web-app>
```

Proxy Servlet Parameters

The parameters and default values for the `HttpProxyServlet` are listed below. For a complete description of these parameters, see HTTP Proxy Servlet Parameters and Proxy Servlet Deployment Parameters.

- `Debug: false`
- `DebugConfigInfo: false`
- `DefaultFileName: null`
- `FileCaching: true`
- `KeepAliveEnabled: true`
- `KeepAliveSecs: 20`
- `MaxPostSize: -1`
- `PathPrepend: null`
- `PathTrim: null`
- `TrimExt: null`
- `SecureProxy: false`
- `WebLogicHost: none`
- `WebLogicPort: none`
- `WLCookieName: JSESSIONID`
- `WLIOTimeoutSecs: 300`
- `WLLogFile: null`
- `WLProxyPassThrough: false`
- `WLProxySSL: false`
- `WLProxySSLPassThrough: false`

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 *Administering Clusters for Oracle WebLogic Server*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

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 *Administering Clusters for Oracle WebLogic Server*.

Referencing a Servlet in a Web Application

The URL used to reference a servlet in a Web application is constructed with a certain pattern.

`https://myHostName:port/myContextPath/myRequest?name=value`

The components of this URL are defined as follows:

- `myHostName`—The DNS name mapped to the Web Server defined in the WebLogic Remote 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.
- `name=value`—Optional query parameters begin with `?`. Separate additional parameters with `&`. Path information, when present, follows the servlet path after a `/` and precedes the query string.

URL Pattern Matching

WebLogic Server provides the user with the ability to implement a URL matching utility which does not conform to the Jakarta EE 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();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  public Object[] values();
  public String[] keys();
}
```

The SimpleApacheURLMatchMap Utility

The included `SimpleApacheURLMatchMap` utility is not Jakarta EE 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.

See url-match-map.

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

> **Note**
>
> **Compatibility guidance:** `weblogic.servlet.http.AbstractAsyncServlet` and the Future Response Servlet are retained for existing applications. For new applications, use the standard asynchronous processing model in Jakarta Servlet 6.1.

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.

Example 9-3 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) {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
            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.

Example 9-4 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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.

Example 9-5 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

> **Note**
>
> **Compatibility guidance:** `weblogic.servlet.http.AbstractAsyncServlet` and `weblogic.servlet.FutureResponseServlet.java` are retained for existing applications. For new applications, use the standard asynchronous processing model in Jakarta Servlet 6.1.

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.

Example 9-6 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{
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  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();
    }
  }
}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

10

Using Sessions and Session Persistence

Learn how to set up and use HTTP sessions and session persistence in WebLogic Server. This chapter includes the following sections:

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.

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.

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`:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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. See session-descriptor.
- Set the `session-timeout` element in the Jakarta EE standard Web application deployment descriptor, `web.xml`.

If you want to synchronize the expiration of all sessions that share the same session ID, then enable the `SynchronizedSessionTimeoutEnabled` attribute in either `WebAppContainerMBean` or `ServerTemplateMBean`. If the attribute is configured in both MBeans, the value set in `ServerTemplateMBean` overrides `WebAppContainerMBean`.

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.

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 session-descriptor.

Logging Out

`session.invalidate()` ends the current `HttpSession` and makes its attributes unavailable. It does not, by itself, guarantee logout from container-managed authentication or single sign-on across applications. When the application uses Servlet container-managed authentication, call `request.logout()` as part of the logout flow and then invalidate the application session. Follow the WebLogic Server single-sign-on procedure when credentials are shared across applications.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

The session descriptor configuration that you specify at the application level overrides any session descriptor configuration that is specified at the individual Web application level. 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="UTF-8"?>

<weblogic-application xmlns="http://xmlns.oracle.com/weblogic/weblogic-application">
   ...
 <session-descriptor>
     <persistent-store-type>memory</persistent-store-type>
     <sharing-enabled>true</sharing-enabled>
     ...
 </session-descriptor>
...
</weblogic-application>
```

Limiting Number of Concurrent Requests for a Session

The `weblogic.http.session.maxConcurrentRequest` property limits the number of concurrent requests for a session. If the number of concurrent requests for a given session exceeds the specified value, the servlet container will start rejecting requests. By default, this property is set to `-1`, which indicates the servlet container does not impose any restrictions.

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.

The following session persistence implementations are supported:

- Memory (single-server, non-replicated)
- File system persistence
- JDBC persistence
- Cookie-based session persistence
- In-memory replication using either WebLogic Server clusters or Coherence clusters

The first four are discussed here; in-memory replication is discussed in HTTP Session State Replication in *Administering Clusters for Oracle WebLogic Server*. For detailed information on using Coherence for session state replication, see *Administering HTTP Session Management with Oracle Coherence*Web*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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" in session-descriptor.

  > **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" in session-descriptor.

  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 session-descriptor.

> **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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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" in session-descriptor.
- Set the directory where WebLogic Server stores the sessions. See "persistent-store-dir" in session-descriptor.

> **Note**
>
> You must create this directory 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 life time 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:

- Set the `persistent-store-type` parameter in the `session-descriptor` element in the `weblogic.xml` deployment descriptor file to `jdbc`. See session-descriptor.
- Set a JDBC connection pool to be used for persistence storage with the `persistent-store-pool` or `persistent-data-source-jndi-name` 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 Remote Console. See session-descriptor.

  With asynchronous JDBC persistence for HTTP sessions in an application or Web application, the `persistent-store-pool` parameter is ignored. To set a JDBC connection pool for `async-jdbc`-based persistence, you must specify the `persistent-data-source-jndi-name` parameter in the `session-desciptor` element in the `weblogic.xml` deployment descriptor file. See session-descriptor.
- 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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 10-1 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`
`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.

Example 10-1 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) );
```

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.

Example 10-2 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 DB2, use the following SQL statement to create the `wl_servlet_sessions` table. Modify the SQL statement for use with your DBMS.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Example 10-3 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.

Example 10-4 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                      CHAR(1)                          null  ,
WL_CREATE_TIME                 decimal(16,0)                     null  ,
WL_IS_VALID                    CHAR(1)                          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 session-descriptor.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 *jakarta.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:

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

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. When URL rewriting is enabled and the application encodes its response URLs, it can provide session tracking for clients that do 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 address and finds the appropriate `HttpSession` when your servlet calls the `getSession()` method.

> **Note**
>
> **Security warning:** Avoid URL rewriting for authenticated or sensitive applications. A session identifier in a URL can be exposed through browser history, access logs, referrer headers, bookmarks, screenshots, and copied links. Prefer secure session cookies unless a documented legacy-client requirement prevents their use.

Enable URL rewriting in WebLogic Server by setting the `url-rewriting-enabled` parameter to `true` in the WebLogic-specific deployment descriptor, `weblogic.xml`, under the `session-descriptor` element. The default value is `false`. See session-descriptor.

Coding Guidelines for URL Rewriting

Here are general guidelines for supporting URL rewriting.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 WebLogic Server URL-rewriting mechanism uses a semicolon ";" as the delimiter, our URL rewriting 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 WebLogic Remote Console at **Server** > **Protocols** > **HTTP**. 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" in session-descriptor.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> If the `id-length` subelement of the `session-descriptor` element of the WebLogic Server-specific deployment descriptor, `weblogic.xml`, contains a value of less than 32, WebLogic Server automatically increases the value to 32 and displays the following message:
>
> ```
> The IDLength is too short. It is not secure. WLS will raise the length to 32.
> ```

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 Jakarta 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);
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 https://jakarta.ee/specifications/servlet/5.0/apidocs/jakarta/servlet/http/httpsession.

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 user name and password.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> 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. See Using URL Rewriting Instead of Cookies.

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 Jakarta EE 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()
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
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

For the distinction between ending an `HttpSession` and logging out from container-managed authentication or single sign-on, see Logging Out.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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. See Configuring WebLogic Server Session Cookies.

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.

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 session-descriptor.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> The CISCO Local Director load balancer expects a question mark "?" delimiter for URL rewriting. Because the WebLogic Server URL-rewriting mechanism uses a semicolon ";" as the delimiter, our URL rewriting 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 session-descriptor.

The minimum value is 32 bytes; the default value is 52 bytes; the maximum value is `Integer.MAX_VALUE`. (See the note in URL Rewriting and Wireless Access Protocol (WAP)).

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. See the `cache-size` property, at Configuring Session Persistence.
- 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. See *Administering Clusters for Oracle WebLogic Server*.
- 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. See Configuring Session Persistence.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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`.

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 Configuring Session Persistence.)

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.

Configuring Session Persistence

For details about setting up persistent sessions, see Configuring Session Persistence.

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 session-descriptor.

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 Jakarta EE 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
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

11

Application Events and Event Listener Classes

Learn about Web application events and event listener classes.
This chapter includes the following sections:

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.)
4. At run time, 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

Examine a listing of 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 11-1 Servlet Context Events

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

HTTP Session Events

HTTP session events report session creation and destruction, activation and passivation, and session-attribute changes.

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 11-2 HTTP Session Events

Type of Event	Interface	Method
An HTTP session is created or destroyed.	`jakarta.servlet.http.HttpSessionListener`	`sessionCreated()`, `sessionDestroyed()`
An HTTP session is activated or passivated.	`jakarta.servlet.http.HttpSessionActivationListener`	`sessionDidActivate()`, `sessionWillPassivate()`
A session attribute is added, removed, or replaced.	`jakarta.servlet.http.HttpSessionAttributeListener`	`attributeAdded()`, `attributeRemoved()`, `attributeReplaced()`

> **Note**
>
> The Jakarta Servlet 6.1 specification also contains the `jakarta.servlet.http.HttpSessionBindingListener` and the `jakarta.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`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Servlet Request Events

Examine a listing of the types of servlet request events, the interface your event listener class must implement to manage state across the life cycle of servlet requests and the methods invoked when the request events occur.

Table 11-3 Servlet Request Events

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

Configuring an Event Listener Class

Learn how to configure an event listener class.

Use `@WebListener` for annotation-based registration:

```
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;

@WebListener
public class MyContextListener implements ServletContextListener {
  ...
}
```

Use the `listener` element in `web.xml` when descriptor-based registration or metadata control is required.

To configure an event listener class in `web.xml`:

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<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.

Writing an Event Listener Class

Learn how to write 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 or HTTP Session Events. See Templates for Event Listener Classes 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 Jakarta EE API Reference (Javadocs) at `https://docs.oracle.com/en/java/javase/17/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:

   - `jakarta.servlet.http.HttpSessionEvent`—provides access to the HTTP session object
   - `jakarta.servlet.ServletContextEvent`—provides access to the servlet context object.
   - `jakarta.servlet.ServletContextAttributeEvent`—provides access to servlet context and its attributes
   - `jakarta.servlet.http.HttpSessionBindingEvent`—provides access to an HTTP session and its attributes

Templates for Event Listener Classes

Examine examples that provide some basic templates for event listener classes.

Servlet Context Event Listener Class Example

```
package myApp;
import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
public final class MyContextListenerClass implements
   ServletContextListener {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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 jakarta.servlet.http.HttpSessionAttributeListener;
import jakarta.servlet.http.HttpSessionBindingEvent;

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

- Jakarta Servlet 6.1 at https://jakarta.ee/specifications/servlet/6.1/
- Jakarta EE 11 API at https://jakarta.ee/specifications/platform/11/apidocs/

Copyright © 2007, 2026, Oracle and/or its affiliates.

12

WebLogic JSP Reference

Review reference information for writing WebLogic JavaServer Pages (JSPs).
This chapter includes the following sections:

JSP Tags

Review the basic tags that you can use in a JSP page.

The following table describes the tags. Each shorthand tag has an XML equivalent.

Table 12-1 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.

`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.

`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.

`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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 12-1 (Cont.) 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 2.2 specification. See Actions .
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.

Defining JSP Versions

WebLogic Server supports JSP 3.0. Changes between JSP 3.0 and JSP 2.1 (JSR 245) include that the API has moved from the `jakarta.servlet.jsp` package to the `jakarta.servlet.jsp` package.

For information about JSP 3.0, see https://jakarta.ee/specifications/pages/3.0/jakarta-server-pages-spec-3.0.

JSP 2.3 was a maintenance release for JSP 2.1 (JSR 245). Because JSP 2.1 imported some new features, the same syntax could hold different meanings between JSP 2.1 and JSP 2.0, so the JSP version must be defined to attain the expected behavior. For example:

- `<%@ page deferredSyntaxAllowedAsLiteral="true" %>` is not allowed in JSP 2.0.
- `# {expr}` is valid in JSP 2.0 template text, but is invalid in JSP 2.1 by default.

Rules for Defining a JSP File Version

Since there is no explicit method of specifying a JSP page's version, its version is eventually determined by the Web application version, as follows:

- If `<jsp:root>` appears in a JSP document, its attribute version value will determine that JSP document's version; otherwise, the Web application version will determine it.
- If the Web application version is determining the JSP version, then 2.5 indicates the version is JSP 2.1 and 2.4 means the version is JSP 2.0.
- If a JSP document contains `<jsp:root>`, and if Web application version is 2.4, the `<jsp:root>` version must not be higher than 2.0. However, if the Web application version is 2.5, then the `<jsp:root>` version could be less than 2.1.
- All Referred JSP tag versions must not be higher than current JSP file's version.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Rules for Defining a Tag File Version

All JSP tag file versions are defined by the version of the tag library they belong to.

- Since an implicit tag library will be created for each directory, including tag files, the implicit tag library's version is 2.0 by default. However, the version can be configured by the `implicit.tld` file in same directory in JSP 2.1.
- A `.tagx` file's `<jsp:root>` attribute version value must be same as the tag file's version.
- All Referred JSP tag versions must not be higher than current tag file's version.

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 specification. The JSP API is described in the Javadocs available at https://jakarta.ee/specifications/platform/11/jakarta-platform-spec-11.0.

> **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.

Table 12-2 Reserved Words for Implicit Objects

Reserved Word	Description
`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`	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. **Note:** 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 `jakarta.servlet.jsp.JspWriter` uses `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 12-2 (Cont.) Reserved Words for Implicit Objects

Reserved Word	Description

`out`	An instance of `jakarta.servlet.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`	Represents a `jakarta.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`	Represents a `jakarta.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 2.1 specification states that if the session directive is set to `false`, then using the `session` keyword results in a fatal translation time error.
`application`	Represents a `jakarta.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`	Represents a `jakarta.servlet.ServletConfig` object and provides access to the servlet instance initialization parameters.
`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 />`

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 *Developing JSP Tag Extensions for Oracle WebLogic Server*.

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.

> **Note**
>
> **Compatibility guidance:** JSP declarations, scriptlets, and expression elements are documented for existing pages and for reference. For new pages, use Expression Language and Jakarta Standard Tag Library (JSTL) 3.0, and place Java logic in controllers, servlets, or managed application components.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Compatibility Example: JSP with Embedded Java 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.
- Encode untrusted data for the output context before returning it. Using the wrong encoder can leave the application vulnerable to cross-site scripting. See Securing User-Supplied Data in JSPs.
- The Java tag is an *inline* tag; it does not force a new paragraph.

Expressions

Learn how 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 supplied. Encode untrusted data for the output context before returning it. Using the wrong encoder can leave the application vulnerable to cross-site scripting. See Securing User-Supplied Data in JSPs.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Compatibility Example: JSP with Embedded Java

This compatibility 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 shown in the following figure.

Figure 12-1 Compiled JSP with HTML and Embedded Java



Copyright © 2007, 2026, Oracle and/or its affiliates.

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"
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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:

Table 12-3 Defining the Scope attribute of a JavaBean Object

Scope	Description
`page`	This is the default scope for a JavaBean, which stores the object in the `jakarta.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 http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 JSP expression language 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}` or `${expr}`.

The `#{expr}` syntax refers to deferred expressions introduced in JSP EL 2.1. Expressions delimited by "`#{}`" use "deferred evaluation" because the expression is not evaluated until its value is needed by the system, and so can be processed by the underlying mechanism at the appropriate moment within its life cycle. Whereas, expressions delimited by "`${}`" use "immediate evaluation" because the expression is compiled when the JSP page is compiled and it is executed when the JSP page is executed. The deferred expression includes deferred `ValueExpression` and deferred `MethodExpression`. The `${expr}` syntax is supported in JSP EL 2.1.

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.

WebLogic Server now supports EL 4.0. For more information on the JSP expression language, see https://jakarta.ee/specifications/expression-language/4.0/.

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 use-cases for expressions in attribute values:

- The attribute value contains a single expression construct of either `<some:tag value="${expr}"/>` or `<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 "Type Conversion," at https://jakarta.ee/specifications/expression-language/4.0/jakarta-expression-language-spec-4.0#type-conversion.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- The attribute value contains one or more expressions separated or surrounded by text of either: `<some:tag value="some${expr}${expr}text${expr}"/>` or `<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 attribute's expected type according to the type conversion rules described in "Type Conversion," at https://jakarta.ee/specifications/expression-language/4.0/jakarta-expression-language-spec-4.0#type-conversion.
- 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 "Type Conversion," at https://jakarta.ee/specifications/expression-language/4.0/jakarta-expression-language-spec-4.0#type-conversion.

> **Note**
>
> These rules are equivalent to the JSP 2.1 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 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}" />
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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`—Represents the `pageContext` object.
- `pageScope`—Represents a Map that maps page-scoped attribute names to their values.
- `requestScope`—Represents a Map that maps request-scoped attribute names to their values.
- `sessionScope`—Represents a Map that maps session-scoped attribute names to their values.
- `applicationScope`—Represents a Map that maps application-scoped attribute names to their values.
- `param`—Represents a Map that maps parameter names to a single String parameter value (obtained by calling `ServletRequest.getParameter(String name)`).
- `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`—Represents a Map that maps header names to a single String header value (obtained by calling `ServletRequest.getHeader(string name)`).
- `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`—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`—Represents a Map that maps context initialization parameter names to their String parameter value (obtained by calling `ServletRequest.getInitParameter(String name)`).

Table 12-4 shows some examples of using these implicit objects:

Table 12-4 Example Uses of Implicit Objects

Expression	Description
`${pageContext.request.requestURI}`	The request's URI (obtained from `HttpServletRequest`)
`${sessionScope.profile}`	The session-scoped attribute named profile (`null` if not found)

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 12-4 (Cont.) Example Uses of Implicit Objects

Expression	Description
`${param.productId}`	The String value of the `productId` parameter (`null` if not found).
`${paramValues.productId}`	The `String[]` containing all values of the `productId` parameter (`null` if not found).

JSP Expression Language Literals and Operators

Learn about JSP EL 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 Collected Syntax in the EL specification.
- Floating point - As defined by the `FloatingPointLiteral` construct in Collected Syntax in the EL 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 []

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 Operators in the EL 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

- null
- div
- mod

> **Note**
>
> 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.

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 Resolution of Variables and their Properties of the JSP 3.0 specification.

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 Example 12-1 prompts a user to enter a string, assigns the string to a parameter named `userInput`, and then uses the `<%= jakarta.servlet.ServletRequest.getParameter("userInput")%>` expression to return the data to the browser.

Example 12-1 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:
    <%= jakarta.servlet.ServletRequest.getParameter("userInput")%>
  </body>
</html>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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. See *Cross Site Scripting Prevention Cheat Sheet* on the Open Web Application Security Project (OWASP) website at https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.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 12-5. 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 12-5 HTML Special Characters that Must Be Replaced

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

Compatibility: Using a WebLogic Server Utility Method

> **Note**
>
> **Compatibility guidance:** The `weblogic.servlet.security.Utils.encodeXSS()` method is retained for existing code. It does not replace context-specific encoding.

For existing code that inserts untrusted data into HTML text, WebLogic Server provides `weblogic.servlet.security.Utils.encodeXSS()`. Review each output location separately because HTML attributes, URLs, JavaScript, and CSS can require different encoders. For example:

```
<%
String userInput = request.getParameter("userInput");
%>
<%= userInput == null ? "" :
    weblogic.servlet.security.Utils.encodeXSS(userInput) %>
```

Using Sessions with JSP

Sessions in WebLogic JSP perform according to the JSP 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 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) 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 at http://www.oracle.com/technetwork/java/index-jsp-141438.html.

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 http://www.oracle.com/technetwork/

Copyright © 2007, 2026, Oracle and/or its affiliates.

java/index-jsp-141438.html. 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 standalone WebLogic JSP compiler, weblogic.jspc, is deprecated. Use weblogic.appc or the current application build process to precompile and validate a WAR or EAR.

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 12-6 JSP Compiler Options

Option	Description
`-classpath`	Add a list (separated by semi-colons on Windows 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 `jdt`. 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.
`-compressHtmlTemplate`	Compress the HTML in the JSP template blocks to improve run-time performance. If the JSP's HTML template block contains the `<pre>` tag, do not enable this option.
`-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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 12-6 (Cont.) JSP Compiler Options

Option	Description
`-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.
`-optimizeJavaExpression`	Optimize Java expressions to improve run-time performance.
`-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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

13

Filters

Learn how to use Java classes known as filters in WebLogic Web applications.
This chapter includes the following sections:

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 Jakarta servlets, Jakarta Server 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.

A filter applies cross-cutting request or response behavior to one or more resources without duplicating that behavior in each servlet or page. Common uses include authentication integration, request logging, request normalization, security headers, and response transformation. Keep filter chains small and measure filters that buffer or transform response bodies, because they can increase memory use and latency.

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 `jakarta.servlet.ServletRequest` and `jakarta.servlet.ServletResponse` objects that provide access to the HTTP request and response, and a `jakarta.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

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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 `jakarta.servlet.Filter` interface.

See https://jakarta.ee/specifications/platform/11/apidocs/jakarta/servlet/filter. 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 Jakarta EE javadocs for `jakarta.servlet.FilterConfig` at https://jakarta.ee/specifications/platform/11/apidocs/jakarta/servlet/filterconfig.

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

Configure a filter with `@WebFilter` when annotation-based configuration is appropriate. Use `web.xml` when the application requires descriptor-based mapping, dispatcher configuration, or explicit chain ordering.

```
import jakarta.servlet.Filter;
import jakarta.servlet.annotation.WebFilter;

@WebFilter(urlPatterns = {"/protected/*"})
public class RequestFilter implements Filter {
  ...
}
```

The following sections describe descriptor-based filter declarations and mappings.

Configuring a Filter

To configure a filter:

1. Open the `web.xml` deployment descriptor in a text editor or an XML editor. See Web Application Developer Tools. 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta Servlet 6.1 specification at https://jakarta.ee/specifications/servlet/6.1/. For example, the following `filter-mapping` maps `myFilter` to requests that contain `/myPattern/`.

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

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

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

5. To create a chain of filters, specify multiple filter mappings. See Configuring a Chain of Filters.

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 `jakarta.servlet.http.HttpServletResponseWrapper` and `jakarta.servlet.http.HttpServletRequestWrapper` at https://jakarta.ee/specifications/platform/11/apidocs/jakarta/servlet/http/package-summary.html.

Additional Resources

- Servlet 6.1 specification at https://jakarta.ee/specifications/servlet/6.1/
- Jakarta EE API Reference (Javadocs) at https://jakarta.ee/specifications/platform/11/apidocs/

Copyright © 2007, 2026, Oracle and/or its affiliates.

14

Using WebLogic JSP Form Validation Tags

Learn how to use WebLogic JavaServer Pages (JSP) form validation tags in WebLogic Server. This chapter includes the following sections:

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

Learn about 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).

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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.

  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.

  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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

<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. See Using a Custom Validator Class.

  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"
    ```

    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

Examine the steps for using 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).

Copyright © 2007, 2026, Oracle and/or its affiliates.

c. Define an HTML form using the <wl:form> JSP tag that is included with the supplied tag library. See <wl:form> and Creating HTML Forms Using the <wl:form> Tag. 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.

2. Add <wl:validator> tags. For the syntax of the tags, see <wl:validator>. 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.

3. 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.

4. 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

Learn how to create 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<p> <input type="submit" value="Submit Form">  </p>
</wl:form>
```

For information on the syntax of this tag see <wl:form>.

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, re-display previously entered values so that the user does not have to fill out the entire form again. Encode each untrusted value with the Jakarta Standard Tag Library (JSTL) 3.0 core tag before placing it in the HTML `<input>` element.

Re-Displaying a Value Using the <input> Tag

You can use the `jakarta.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.

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://attic.apache.org/projects/jakarta-taglibs.html.

To use the Apache tag library in your JSP:

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Learn how to use your own validator class.

1. Write a Java class that extends the `weblogicx.jsp.tags.validators.CustomizableAdapter` abstract class. See Extending the CustomizableAdapter Class.
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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Sample User-Written Validator Class

Example 14-1 Example of a User-written Validator Class

```
import weblogicx.jsp.tags.validators.CustomizableAdapter;

public class myCustomValidator extends CustomizableAdapter{

    public myCustomValidator(){
super();
    }

    public boolean validate(jakarta.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

Examine sample code that shows the basic structure of a JSP using 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.

Because this sample collects password values, use POST and serve and submit the form only over HTTPS. Do not place credentials in a query string.

Example 14-2 JSP with WebLogic JSP Form Validation Tags

```
<%@ taglib uri="tagl" prefix="wl" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>


<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 )   {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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="POST" action="successPage.jsp">

 User Name: <input type="text" name="username"
                   value="<c:out value='${param.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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

15

Using Custom WebLogic JSP Tags (cache, process, repeat)

Learn how to use three custom JSP tags—`cache`, `repeat`, and `process`—provided with the WebLogic Server distribution.

> **Note**
>
> **Compatibility guidance:** The WebLogic `cache`, `process`, and `repeat` JSP tags are retained for existing applications. For new applications, keep request processing in controllers or services, use Expression Language and Jakarta Standard Tag Library (JSTL) 3.0 for view iteration and conditions, and use the application's supported caching layer instead of introducing JSP scriptlets or JSP-specific flow control.
>
> The compatibility examples in the `process` and `repeat` sections retain the scripting style used by applications built with these proprietary tags. Do not use them as the model for a new JSP page.

This chapter includes the following sections:

Overview of WebLogic Custom JSP Tags

Oracle 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 Jakarta EE 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

> **Note**
>
> **Security warning:** A cached page fragment can contain user-specific or authorization-sensitive output. Include every value that changes the rendered result—such as user identity, role, locale, tenant, and relevant request parameters—in the cache design, or do not cache the fragment. Never share a fragment containing one user's sensitive data with another user.

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.

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 Table 15-1) 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`).

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 15-1 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 post fixing 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 15-1 (Cont.) 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.
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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 15-1 (Cont.) 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.
`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.

Example 15-1 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">
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<!--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 15-2 Process Tag Attributes

Tag Attribute	Required	Description
name	no	Name of a query parameter.
notname	no	Name of a query parameter.
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:

Example 15-2 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
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
 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 15-3 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.

Example 15-3 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

16

Using the WebLogic EJB to JSP Integration Tool

Learn how to use the WebLogic EJB-to-JSP integration tool to create JSP tag libraries that you can use to invoke EJBs in a Jakarta Server Page (JSP) for WebLogic Server. This document assumes at least some familiarity with both EJB and JSP.

> **Note**
>
> **Compatibility guidance:** The EJB-to-JSP Integration Tool supports existing EJB 2.x applications that use home interfaces, remote interfaces, entity beans, and generated JSP tag libraries. It is not the recommended architecture for a new Jakarta EE application. For new development, invoke application services from a controller, servlet, or managed component and render the result with the selected view technology.

This chapter includes the following sections:

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 jakarta.ejb.EJBObject {
  public TradeResult buy(String stockSymbol, int shares);
  public TradeResult sell(String stockSymbol, int shares);
}
```

Existing EJB 2.x/JSP applications can contain EJB invocation code in JSP scriptlets. The EJB-to-JSP tool was designed to replace that scriptlet code with a generated tag library. New applications should keep EJB or service invocation outside the JSP view.

The EJB-to-JSP tool simplifies the EJB invocation process by removing the need for Java code. Instead, 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,

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 or graphical mode.

Use the following command:

```
java weblogic.servlet.ejb2jsp.Main
```

Invoke the graphical tool as follows:

```
java weblogic.servlet.ejb2jsp.gui.Main
```

For all but the simplest EJBs, the graphical tool is preferable.

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, 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta EE 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 Jakarta EE API. Typically, nothing will need to be added to this field.

Troubleshooting

Sometimes, a project fails to build because of errors or conflicts. Read about some of 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 also disable tags for an entire bean.
- `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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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, and such).

EJB Home Methods

EJBs allow 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:
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<%@ 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 a 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, 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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

A

web.xml Deployment Descriptor Elements

Read descriptions of the standard Jakarta EE deployment descriptor elements for WebLogic Server.
With Jakarta EE annotations, the standard `web.xml` deployment descriptor is optional. Jakarta Servlet 6.1 supports annotation-based configuration for servlets, filters, and listeners. A `web.xml` descriptor is optional unless the application requires descriptor-based configuration or metadata. See https://jakarta.ee/specifications/servlet/6.1/ and Servlet Annotation and Dependency Injection.

This appendix includes the following sections:

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="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://
jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         version="5.0">
```

To view the schema for `web.xml`, go to https://jakarta.ee/xml/ns/jakartaee/#9.

context-param

The optional `context-param` element contains the declaration of a Web application's servlet context initialization parameters.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-1 context-parameter Elements

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 > General** page in the WebLogic Remote Console. For all Web applications hosted by server instances in a cluster, on the **Cluster > General** page.

The following table describes the reserved context parameters used by the Web application container, which have been deprecated and have replacements in `weblogic.xml`.

Table A-2 Deprecated context-param Elements

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.
`weblogic.httpd.servlet.reloadCheckSecs`	Defines 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 auth-filter.
`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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-2 (Cont.) Deprecated context-param Elements

Deprecated Parameter	Description	Replacement Element in weblogic.xml
`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. For additional examples of servlet mapping, see Servlet Mapping.

description

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

Table A-3 description Elements

Element	Required/ Optional	Description
`<description>`	Optional	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.

Table A-4 display-name Elements

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

distributable

The distributable element is not used by WebLogic Server.

Table A-5 description Elements

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Table A-6 ejb-local-ref Elements

Element	Required/ Optional	Description
`<description>`	Optional	A text description of the reference.
`<ejb-ref-name>`	Required	Contains the name of an EJB reference. The EJB reference is an entry in the Web application's environment and is relative to the `java:comp/env` context. The name must be unique within the Web application. It is recommended that name is prefixed with `ejb/`. For example: `<ejb-ref-name>ejb/Payroll</ejb-ref-name>`

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

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

`<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>`

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-6 (Cont.) ejb-local-ref Elements

Element	Required/ Optional	Description
<lookup-name>	Optional	The JNDI name to be looked up to resolve a resource reference.

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.

Table A-7 ejb-ref Elements

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. See ejb-reference-description.
<ejb-ref-type>	Required	The expected Java class type of the referenced EJB.
<home>	Required	The fully qualified class name of the EJB home interface.
<remote>	Required	The fully qualified class name of the EJB remote interface.
<ejb-link>	Optional	The <ejb-name> of an EJB in an encompassing Jakarta EE 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.
<lookup-name>	Optional	The JNDI name to be looked up to resolve a resource reference.

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.

Table A-8 env-entry Elements

Element	Required/Optional	Description
<description>	Optional	A textual description.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-8 (Cont.) env-entry Elements

Element	Required/Optional	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`

`<lookup-name>`	Optional	The JNDI name to be looked up to resolve a resource reference.

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.

See Customizing HTTP Error Responses.

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.

Table A-9 error-page Elements

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`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

filter

The `filter` element defines a filter class and its initialization attributes. For more information on filters, see Configuring Filters.

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

Table A-10 filter Elements

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-11 filter-mapping Elements

Element	Required/Optional	Description
<dispatcher>	Optional	Indicates whether filters should be invoked under request dispatcher forward() and include() calls. You can use the <dispatcher> element to indicate for a filter-mapping whether a filter should be applied to various types of requests. Possible values include: • REQUEST • FORWARD • INCLUDE • ERROR • ASYNC If the <dispatcher> element is absent, the filter is applied to requests when the request comes directly from the client. See "Filters and the RequestDispatcher" in the Servlet 4.0 specification at https://jcp.org/en/jsr/detail?id=369.
<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.
<servlet>	Required - or map by <url-pattern>	The name of a servlet which, if called, causes this filter to execute.
<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 4.0 specification.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-12 Icon Elements

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.

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.

Table A-13 jsp-config Elements

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.

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.

Table A-14 taglib Elements

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-14 (Cont.) taglib Elements

Element	Required/Optional	Description
`<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.

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.

Table A-15 jsp-property-group Elements

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 later `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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-15 (Cont.) jsp-property-group Elements

Element	Required/Optional	Description
`<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.
`<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 4.0 specification.

`default-content-type`	Optional	Specifies the default `contentType` property. Valid values are those of the `contentType` page directive. If the page directive does not include a `contentType` attribute, it specifies the default response `contentType`.
`buffer`	Optional	Specifies the default buffering model for `JspWriter`. Valid values are those of the buffer attribute of the page directive. Specifies if buffering should be used for the output to response, and if so, the size of the buffer to use.
`error-on-undeclared-namespace`	Optional	Controls whether an error should be raised for the use of an undeclared tag in a JSP document. If set to true, when an undeclared tag is used in a JSP document, an error must be raised during the translation time. Disabled (false) by default.

listener

Define an application listener using the `listener` element.

Table A-16 listener Elements

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

See Configuring an Event Listener Class.

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.

Table A-17 config

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` 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 *Developing Applications with the WebLogic Security Service*.
`<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 of the Web application is used by default. 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.

form-login-config

This is an element within the login-config.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-18 form-login-config Elements

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.

message-destination-ref

The optional `message-destination-ref` element specifies a reference to a message destination associated with a resource. The logical destination described by this element is mapped to a physical destination in the deployment descriptor.

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

Table A-19 message-destination-ref Elements

Element	Required/Optional	Description
`description`	Optional	Provides a description of the message destination reference.
`message-destination-name`	Required	Specifies a name for a message destination. This name must be unique among the names of message destinations within the deployment descriptor.
`mapped-name`	Optional	Maps this message destination to a "logical" name.
`lookup-name`	Optional	The JNDI name to be looked up to resolve the message destination.
`message-destination-type`	Required	Specifies the type of the destination. The type is specified by the Java interface expected to be implemented by the destination. Must be supplied unless an injection target is specified, in which case the type of the target is used. If both are specified, the type must be assignment compatible with the type of the injection target.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-19 (Cont.) message-destination-ref Elements

Element	Required/Optional	Description
`message-destination-usage`	Optional	Specifies the use of the message destination indicated by the reference. The value indicates whether messages are consumed from the message destination, produced for the destination, or both. Valid values are one of the following: • `Consumes` • `Produces` • `ConsumesProduces` If not specified, `ConsumesProduces` is assumed.
`message-destination-link`	Optional	Links a message destination reference or message-driven bean to a message destination.

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.

Table A-20 mime-mapping Elements

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`.

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>jakarta.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-21 resource-env-ref

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.
`<lookup-name>`	Optional	The JNDI name to be looked up to resolve a resource reference.

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.

Table A-22 resource-ref Elements

Element	Required/Optional	Description
`<description>`	Optional	A text description.
`<res-ref-name>`	Required	The name of the resource used in the JNDI tree. Servlets in the Web application use this name to look up a reference to the resource.
`<res-type>`	Required	The Java type of the resource that corresponds to the reference name. Use the full package name of the Java type.
`<res-auth>`	Required	Used to control the resource sign on for security. If set to `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.
`<res-sharing-scope>`	Optional	Specifies whether connections obtained through the given resource manager connection factory reference can be shared. Valid values: Shareable Unshareable

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-22 (Cont.) resource-ref Elements

Element	Required/Optional	Description
`<lookup-name>`	Optional	The JNDI name to be looked up to resolve a resource reference.

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 Resources Using Roles and Policies for Oracle WebLogic Server*. Also, for more information on WebLogic Security, refer to *Developing Applications with the WebLogic Security Service*.

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

Table A-23 security-constraint Elements

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.
`<user-data-constraint>`	Optional	Defines how the client should communicate with the server. See also user-data-constraint.

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.

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

Table A-24 web-resource-collection Elements

Element	Required/Optional	Description
`<web-resource-name>`	Required	The name of this Web resource collection.
`<description>`	Optional	A text description of this security constraint.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-24 (Cont.) web-resource-collection Elements

Element	Required/Optional	Description
`<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.

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.

Table A-25 auth-constraint Elements

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.

user-data-constraint

This is an element within the security-constraint.

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.

Table A-26 user-data-constraint Elements

Element	Required/Optional	Description
`<description>`	Optional	A text description.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-26 (Cont.) user-data-constraint Elements

Element	Required/Optional	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.

security-role

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

Table A-27 security-role Elements

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. See security-role-assignment.

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.

Table A-28 servlet Elements

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-28 (Cont.) servlet Elements

Element	Required/Optional	Description
`<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>`	Optional	The fully-qualified class name of the servlet. As of servlet 3.1, `<servlet-class>` and `<jsp-file>` are optional. Servlet configuration without `<servlet-class>` and `<jsp-file>` is considered preliminary; you should use the programmatical Servlet API to register the servlet dynamically, otherwise, deployment will fail.
`<jsp-file>`	Optional	The full path to a JSP file within the Web application, relative to the Web application root directory. As of servlet 3.1, `<servlet-class>` and `<jsp-file>` are optional. Servlet configuration without `<servlet-class>` and `<jsp-file>` is considered preliminary; you should use the programmatical Servlet API to register the servlet dynamically, otherwise, deployment will fail.
`<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.
`<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.

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Table A-29 icon Elements

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.

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 `jakarta.servlet.ServletConfig.getInitParameter()` method.

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

Table A-30 init-param Elements

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.

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.

Table A-31 security-role-ref Elements

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-31 (Cont.) security-role-ref Elements

Element	Required/Optional	Description
`<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.

Table A-32 servlet-mapping Elements

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-32 (Cont.) servlet-mapping Elements

Element	Required/Optional	Description

`<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 4.0 specification.
		For additional examples of servlet mapping, see Servlet Mapping.
		Note
		In WebLogic Server 15.1.1.0.0 if the same `<url-pattern>` is specified in both the `@WebServlet` annotation and the `web.xml`, the deployment fails due to conflicting URL mappings. To resolve this issue define the `<url-pattern>` in only one place, either in the `@WebServlet` annotation or in the `web.xml` file.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table A-33 session-config Elements

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. See session-descriptor.

web-app

The XML schema for the Servlet 4.0 deployment descriptor. WebLogic Server fully supports HTTP servlets as defined at `https://jcp.org/en/jsr/detail?id=369`. However, the `version` attributed must be set to `4.0` in order to enforce 4.0 behavior.

The following table describes the elements you can define within an `web-app` element.

Table A-34 web-app Elements

Element	Required/Optional	Description
`<version>`	Required	All servlet deployment descriptors must indicate the 4.0 version of the schema in order to enforce servlet 4.0 behavior.

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.

See Configuring Welcome Files.

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

Table A-35 welcome-file-list

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

B

weblogic.xml Deployment Descriptor Elements

This is 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.
This appendix includes the following sections, which 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://xmlns.oracle.com/weblogic/weblogic-web-app">
```

To view the schema for `weblogic.xml`, go to `http://xmlns.oracle.com/weblogic/weblogic-web-app/1.9/weblogic-web-app.xsd`.

async-descriptor

Use the `async-descriptor` element to configure the asynchronous processing behavior of Web applications. The following table describes the elements you can define within an `async-descriptor` element.

Table B-1 async-descriptor Elements

Element	Required/Optional	Description
`timeout-secs`	Optional	Sets the time, in seconds, that WebLogic Server waits before timing out an asynchronous job. The default value is 120 seconds. Setting the timeout to `-1` indicates that the asynchronous job never times out.
`timeout-check-interval-secs`	Optional	Sets the time, in seconds, that WebLogic Server waits between doing checks for timed-out jobs. The default value is 30 seconds.

async-work-manager

Use the `async-work-manager` element to specify a Work Manager for asynchronous jobs, including asynchronous dispatches initiated using the `AsyncContext` `dispatch` methods and runnable jobs started using the `AsyncContext` `start` method. If no Work Manager is specified, the asynchronous jobs will be executed in the current request Work Manager.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

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>
```

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>
```

See Mapping IANA Character Sets to Java Character Sets.

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

Table B-2 charset-mapping Elements

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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>
```

See Determining the Encoding of an HTTP Request.

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

Table B-3 input-charset Elements

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.

container-descriptor

The `container-descriptor` element specifies a list of parameters that affect the behavior of the Web application.

access-logging-disabled

The `access-logging-disabled` element defines whether to eliminate access logging of the underlying Web application. Setting this property to `true` improves server throughput by reducing the logging overhead. If the property is not specified or a `false` value is set, application accesses are logged.

allow-all-roles

In the security-constraints elements defined in the `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`.

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

> **Note**
>
> As a best practice, Oracle recommends that you do not enable the `check-auth-on-forward` property.

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, ensures 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.

container-initializer-enabled

The `container-initializer-enabled` element controls whether or not to enable the servlet container initializer.

In Servlet 3.x applications, `ServletContainerInitializer` is enabled by default. For performance considerations, you can explicitly disable the servlet container initializer by configuring the `container-initializer-enabled` element in the `weblogic.xml` deployment descriptor in the targeted Web application. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
    xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-web-app
                        http://xmlns.oracle.com/weblogic/weblogic-web-app/1.9/weblogic-
web-app.xsd">
    ...
    <container-descriptor>
        <container-initializer-enabled>false</container-initializer-enabled>
    </container-descriptor>
     ...
</weblogic-web-app>
```

In pre-servlet 3.x applications, you can explicitly enable the servlet container initializer by setting the `container-initializer-enabled` element in the `weblogic.xml` deployment descriptor to `true`. For example:

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<container-descriptor>
    <container-initializer-enabled>true</container-initializer-enabled>
</container-descriptor>
```

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.

disable-implicit-servlet-mappings

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`.

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`.

gzip-compression

Section Title

The `gzip-compression` element controls GZIP compression support for a specified Web application.

Table B-4 gzip-compression sub-elements

Element	Description	Default Value
`enabled`	Enables GZIP compression for the specified Web application. If set to true, only the current application is affected. If specified, the `weblogic.xml` value overrides the domain-level value.	`false`
`min-content-length`	Specifies the minimum file size to trigger compression for the specified Web application. This element allows you to bypass small-sized resources where compression would not yield a great return but use unnecessary CPUs. If specified, the `weblogic.xml` value overrides the domain-level value.	`2048`

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-4 (Cont.) gzip-compression sub-elements

Element	Description	Default Value
`content-type`	Specifies the type of content to be included in compression. You can specify more than one content type by using separate content-type sub-elements for each type. If specified, the `weblogic.xml` value overrides the domain-level value.	`text/html, text/xml, text/plain`

If the `gzip-compression` element and all of its sub-elements are present, these values override any default values at the domain level. If one of the sub-elements is absent, then the default domain value for that attribute is used.

The following example demonstrates setting the `gzip-compression` element and its sub-elements:

```
<weblogic-web-app>
   <container-descriptor>
      <gzip-compression>
         <enabled>true</enabled>
         <min-content-length>4096</min-content-length>
         <content-type>text/html</content-type>
         <content-type>text/xml</content-type>
      </gzip-compression>
   </container-descriptor>
</weblogic-web-app>
```

(Optional) Enter reference information in this section.

Syntax

(Optional) Enter syntax information here.

Example B-1 Example Title

(Optional) Enter an example to illustrate your reference here.

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`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

langtag-revision

The `langtag-revision` element determines the language tag specification version that the `HttpServletRequest` `getLocale` and `getLocales` methods should obey.

Currently, WebLogic Server supports RFC5646 and RFC3066. If you do not set a value, the `HttpServletRequest` `getLocale` and `getLocales` methods return a language tag for locale according to RFC5646. The value 3066 means that the `HttpServletRequest` `getLocale` and `getLocales` methods return a language tag for locale according to RFC3066. For example, if using RFC3066:

```
<container-descriptor>
  <langtag-revision>3066</langtag-revision>
</container-descriptor>
```

The system property `-Dweblogic.servlet.langtagRevision` can also determine the locale parsing mechanism. However, explicit configuration for the `langtag-revision` element in `weblogic.xml` takes precedence over configuration in `-Dweblogic.servlet.langtagRevision`. If you do not set a value in `weblogic.xml`, then the system property configuration takes effect.

The following table describes the relationship between the `langtag-revision` element in `weblogic.xml`, the system property `-Dweblogic.servlet.langtagRevision`, and RFC3066 behavior.

System Property	`weblogic.xml`	Uses RFC3066 behavior
not set/5646	not set/5646	off
not set/5646	3066	on
3066	not set	on
3066	5646	off
3066	3066	on

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 in Bytes 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 `4000`.

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.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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`.

prefer-application-packages

The `prefer-application-packages` element specifies a list of packages for classes that must always be loaded from the application. See `prefer-application-packages` in *Developing Applications for Oracle WebLogic Server*.

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

<weblogic-web-app
    xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-web-app
                        http://xmlns.oracle.com/weblogic/weblogic-web-app/1.9/weblogic-
web-app.xsd">

     <context-root>FilterWeb</context-root>
     <container-descriptor>
         <prefer-application-packages>
             <package-name>com.oracle.foo</package-name>
         </prefer-application-packages>
     </container-descriptor>
</weblogic-web-app>
```

Note that in order to use `prefer-application-packages` or `prefer-application-resources`, `prefer-web-inf-classes` must be set to false.

prefer-application-resources

The `prefer-application-resources` element specifies a list of resources that must always be loaded from the application, even if the resources are found in the system classloader. See `prefer-application-resources` in *Developing Applications for Oracle WebLogic Server*.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app">
  <container-descriptor>
    <prefer-web-inf-classes>false</prefer-web-inf-classes>
    <prefer-application-packages>
      <package-name>jakarta.faces.*</package-name>
      <package-name>com.sun.faces.*</package-name>
      <package-name>com.bea.faces.*</package-name>
    </prefer-application-packages>

    <prefer-application-resources>
      <resource-name>jakarta.faces.*</resource-name>
      <resource-name>com.sun.faces.*</resource-name>
      <resource-name>com.bea.faces.*</resource-name>
      <resource-name>META-INF/services/jakarta.servlet.ServletContainerInitializer</
resource-name>
      </prefer-application-resources>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  </container-descriptor>
</weblogic-web-app>
```

Note that in order to use `prefer-application-packages` or `prefer-application-resources`, `prefer-web-inf-classes` must be set to false.

prefer-forward-query-string

When `HttpServletRequest.getQueryString()` is invoked in a forwarding request, WebLogic Server returns the queryString sent by the forwarding servlet via RequestDispatcher and the original ones sent by the client.

When the `prefer-forward-query-string` flag is set to `true`, WebLogic Server returns only the forwarded query string, if it is specified. The default value 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`.

> **Note**
>
> Neither `prefer-application-packages` nor `prefer-application-resources` can be specified when `prefer-web-inf-classes` is turned on in `weblogic.xml`.

redirect-with-absolute-url

The `redirect-with-absolute-url` element controls whether the `jakarta.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.

> **Note**
>
> User readable data used in a redirect.

referer-validation

To help mitigate Cross-Site Request Forgery (CSRF) attacks, you can configure validation of the Referer header in incoming HTTP requests.

Checking the Referer is a commonly used method of preventing CSRF on embedded network devices because it does not require any per-user state. This makes Referer a useful method of CSRF prevention when memory is scarce or server-side state doesn't exist. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<container-descriptor>
<referer-validation>NONE</referer-validation>
</container-descriptor>
```

Valid values:

- `NONE`: Disable Referer header validation.
- `LENIENT` (default): The web container blocks requests whose Referer header has an incorrect value. If a requests lacks the header, the web container accepts the request.
- `STRICT`: The web container blocks requests that lack a Referer header.

For example, an authentication request is sent to `http://myhost:myport/myapp/Jj_security_check`:

- If `<referer-validation>NONE</referer-validation>`, then the container will not validate the Referer header.
- If `<referer-validation>LENIENT</referer-validation>` and there is a Referer header in this request, then the container will check the host and port of Referer URL.
  - If they are "`myhost`" and "`myport`", then this Referer header is valid.
  - If either the host or port of the Referer URL is different from the actual URL, for example, ”`myhost1`”, then this Referer header is invalid.
  - If there is no Referer header in this request, then the container will not validate it.
- If `<referer-validation>STRICT</referer-validation>` and there is a Referer header in this request, the container will check the host and port of the Referer URL.
  - If they are "`myhost`" and "`myport`", then this Referer header is valid.
  - If either the host or port of the Referer URL is different from the actual URL, for example, ”`myhost1`”, then this Referer header is invalid.
  - If there is no Referer header in this request, then the validation will fail.

> **Note**
>
> The web container also will consider the IP address. For example, if `192.168.226.129` is mapped to "`myhost`", then it is valid if the host of the Referer URL is "`192.168.226.129`".

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.

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<container-descriptor>
    <require-admin-traffic>true</require-admin-traffic>
</container-descriptor>
```

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 never reload. This is the default value in a production environment.
- The value `0` means always reload.
- The value `1` means reload every second. This is the default value in a development environment.

> **Note**
>
> If the resource is a JSP, and if `page-check-seconds`is specified in the `jsp-descriptor` element, the `page-check-seconds` value is used to determine reload time for the JSP file.

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`.

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.

session-monitoring-enabled

The `session-monitoring-enabled` element, if set to true, allows run-time MBeans to be created for sessions. When set to `false`, the default value, run-time MBeans are not created.

show-archived-real-path-enabled

The `show-archived-real-path-enabled` element specifies the behavior of `getRealPath()` for archived Web applications.

Copyright © 2007, 2026, Oracle and/or its affiliates.

When set to `true`, `getRealPath()` returns the canonical path of the resource files.

If the `show-archived-real-path-enabled` element is set to `false`, the servlet container will return the real path of files in archived Web applications as `null`.

The default value is `false`.

single-threaded-servlet-pool-size

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

> **Note**
>
> This element is retained only for compatibility with applications that used the removed Servlet `SingleThreadModel` API.

temp-dir

The `temp-dir` element specifies the location of the temporary directory for the Web application, as returned by the `"jakarta.servlet.context.tempDir"` attribute.

context-root

The `context-root` element defines the context root of this standalone Web application. If the Web application is part of an EAR, not standalone, 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:

- Check `context-root` and `web-uri` in `application.xml` for context root; if found, use as Web application's context root.
- 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.
- If context root is not defined in `weblogic.xml` or `application.xml`, then infer the context path from the URI, giving it the name of the value defined in the URI minus the WAR suffix. For instance, a URI `MyWebApp.war` would be named `MyWebApp`.

> **Note**
>
> The `context-root` element cannot be set for individual Web applications in EAR libraries. It can only bet set for Web application libraries.

Copyright © 2007, 2026, Oracle and/or its affiliates.

description

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

ejb-reference-description

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

Table B-5 ejb-reference-description Elements

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.

fast-swap

The following table describes the elements you can define within a `fast-swap` element.

For more information about FastSwap Deployment, see Using FastSwap Deployment to Minimize Redeployment in *Deploying Applications to Oracle WebLogic Server*.

Table B-6 fast-swap Elements

Element	Required/Optional	Description
`enabled`	Optional	Set to `true` to enable FastSwap deployment in your application.
`refresh-interval`	Optional	FastSwap checks for changes in application classes when an incoming HTTP request is received. Subsequent HTTP requests arriving within the `refresh-interval` seconds will not trigger a check for changes. The first HTTP request arriving after the `refresh-interval` seconds have passed, will cause FastSwap to perform a class-change check again.
`redefinition-task-limit`	Optional	FastSwap class redefinitions are performed asynchronously by redefinition tasks. They can be controlled and inspected using JMX interfaces. Specifies the number of redefinition tasks that will be retained by the FastSwap system. If the number of tasks exceeds this limit, older tasks are automatically removed.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-7 jsp-descriptor Elements

Element	Default Value	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.

`strict-stale-check`	`true`	Applies to exploded WARs only.
		Checks for updated JSP files, in other words, whether the timestamp on the file is later (more recent) than the one in the build. Only newer files can replace older ones.
		When set to false, just checks whether the timestamp has changed. If so, the file is replaced.
		`<?xml version="1.0" encoding="UTF-8"?>`
		`<weblogic-web-app xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app">`
		`<jsp-descriptor>`
		`<strict-stale-check>false`
		`</strict-stale-check>`
		`</jsp-descriptor>`
		`</weblogic-web-app>`

`precompile`	`false`	When set to true, WebLogic Server automatically precompiles all JSPs when the Web application is deployed or re-deployed or when starting WebLogic Server.
`precompile-continue`	`false`	When set to true, WebLogic Server continues precompiling all 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.
`debug`	`false`	When set to `true`, WebLogic Server enables the debugging feature of the JSP compiler. The default value is `false`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-7 (Cont.) jsp-descriptor Elements

Element	Default Value	Description
`verbose`	`false`	For production mode, the default value is `false`, but for development mode, the default value is `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.
`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. See Backward Compatibility Flags.
`encoding`	`UTF-8 for JSP and JSPX pages`	Specifies the default character set used in the JSP page. Use standard Java character set names (see `https://docs.oracle.com/en/java/javase/17/intl/supported-encodings.html`). 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 run-time expression values in the name attribute of the `jsp:param` tag. It is set to `false` by default.
`optimize-java-expression`	`false`	When set to true, the JSP compiler optimizes Java expressions to improve run-time performance.
`compress-html-template`	`false`	When set to true, compresses the HTML in the JSP template blocks to improve run-time performance. If the JSP's HTML template block contains the `<pre>` HTML tag, do not enable this feature.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Table B-8 library-ref Elements

Element	Required/Optional	Description
`library-name`	Required	Provides the library name for the library module reference. The default value is `null`.
`specification-version`	Optional	Provides the specification version for the library module reference. The default value is `0`. (This is a float.)
`implementation-version`	Optional	Provides the implementation version for the library module reference. The default value is `null`.
`exact-match`	Optional	The default value is `false`.

logging

The `logging` element is a sub-element of the `weblogic-web-app` element. You can define the following elements within the `logging` element.

Table B-9 logging Elements

Element	Required/Optional	Description
`log-filename`	Required	Specifies the name of the log file. The full address of the filename is required.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-9 (Cont.) logging Elements

Element	Required/Optional	Description
`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`
`rotation-type`	Optional	Sets the file rotation type. Values are `bySize, byTIme, none` • `bySize`—When the log file reaches the size that you specify in `file-size-limit`, the server renames the file as `FileName.n`. • `byTIme`—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`
`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

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-9 (Cont.) logging Elements

Element	Required/Optional	Description
`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`
`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

ready-registration

To use the ReadyApp framework, register a WAR-based application with the framework by adding the following code to the application's WebLogic deployment descriptor (`META-INF\weblogic-application.xml`):

```
<wls:ready-registration>true</wls:ready-registration>
```

When the application starts, the state of the application is set to NOT READY.

> **Note**
>
> The prefix `wls:` may or may not be required depending on the contents of the `weblogic-application.xml` file. If the rest of the tags do not have the prefix, you can ignore the prefix.

See Configuring the ReadyApp Framework with EAR or WAR-based Applications in Deploying Applications to Oracle WebLogic Server.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Table B-10 resource-description Elements

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.

Table B-11 resource-env-description Elements

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.

run-as-role-assignment

The `run-as-role-assignment` element maps a `run-as` role name (a sub-element 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-12 run-as-role-assignment Elements

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.

security-permission

The `security-permission` element specifies a single security permission based on the security policy file syntax. Refer to Default Policy Implementation and Policy File Syntax for the implementation of the security permission specification.

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.

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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> 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.

Table B-13 security-role-assignment Elements

Element	Required/Optional	Description
`role-name`	Required	Specifies the name of a security role.
`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 *Administering Security for Oracle WebLogic Server*.
`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.

If you do not define a `security-role-assignment` element and its sub-elements, 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.

service-reference-description

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

Table B-14 service-reference-description Elements

Element	Required/Optional	Description
`service-ref-name`		
`wsdl-url`		

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-14 (Cont.) service-reference-description Elements

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`

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.

Table B-15 servlet-descriptor Elements

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 Work Manager by identifying the Work Manager name. This setting overrides the Web application-level dispatch policy defined by `wl-dispatch-policy`.

session-descriptor

The following table describes the elements you can define within a `session-descriptor` element to define parameters for servlet sessions.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-16 session-descriptor

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.
`invalidate-on-relogin`	`false`	Sets whether the container must invalidate the current session if the currently logged-in user switches to a different user name (which is valid in the security realm) and attempts to log in again. If the value of this parameter is set to `true`, the current session is invalidated if the user attempts to log in again using a different user name.
`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`.
`id-length`	`52`	Sets the size of the session ID. The minimum value is 32 bytes and the maximum value is `Integer.MAX_VALUE`. **Note:** If a value lower than 32 bytes is set, WebLogic Server automatically raises the value to 32 and displays the following message: `The IDLength is too short. It is not secure. WLS will raise the length to 32.`
`tracking-enabled`	`true`	Enables session tracking between HTTP requests.
`cache-size`	`1024`	Sets the cache size for JDBC and file-persistent sessions.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-16 (Cont.) session-descriptor

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 grows 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); any negative value works as the same as `-1`.
`max-save-post-size`	`4096`	Sets the maximum size, in bytes, of the POST data that the container saves/buffers during FORM authentication. The default value is `4096` bytes.
`save-post-timeout-secs`	`40`	Defines the timeout, in seconds, for the session that saved/buffered POST data. For FORM authentication, POST data is saved in a session while the user is redirected to the login form. The default value is `40` seconds. If the value of the `save-post-timeout-secs` element is less than the value of the `timeout-secs` element, then session invalidation may occur during user operations. In this scenario, increase the value of `save-post-timeout-secs` to match the `timeout-secs` value or to an acceptable value, according to your needs.
`save-post-timeout-interval-secs`	`20`	Sets the invalidation trigger interval, in seconds, for saving POST data in a session. The default value is `20` seconds.
`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`	`/`	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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-16 (Cont.) session-descriptor

Element Name	Default Value	Value
`cookie-domain`	`null`	Specifies the domain for which the cookie is valid. For example, setting `cookie-domain` to `.example.com` returns cookies to any server in the `*.example.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. See `Cookie.setDomain()` in the Servlet specification.
`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 Web sites 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. This value can be set as any integer; the default value is -1 (unlimited). For more information about cookies, see Using Sessions and Session Persistence.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-16 (Cont.) session-descriptor

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. • `async-replicated`—Enables asynchronous session replication in an application or Web application. See Asynchronous HTTP Session Replication in *Tuning Performance of Oracle WebLogic Server*. • `async-replicated-if-clustered`—Enables asynchronous session replication in an application or Web application when deployed to a cluster environment. If deployed to a single server environment, then the session persistence/replication defaults to in-memory. This allows testing on a single server without deployment errors. • `file`—Uses file-based persistence (See also session-descriptor). • `async-jdbc`—Enables asynchronous JDBC persistence for HTTP sessions in an application or Web application. See Configuring Session Persistence. • `jdbc`—Uses a database to store persistent sessions. (see also session-descriptor). • `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. See Using Cookie-Based Session Persistence.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-16 (Cont.) session-descriptor

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-data-source-jndi-name`	`None`	Specifies the data source JNDI name of a JDBC connection to be used for `jdbc`- and `async-jdbc`-based persistence (see `persistent-store-type` above). For `async-jdbc`-based persistence, you must specify the `persistent-data-source-jndi-name` parameter to configure 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`	`wl_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.
`url-rewriting-enabled`	`false`	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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-16 (Cont.) session-descriptor

Element Name	Default Value	Value
`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.
`monitoring-attribute-name`		Configures the monitoring ID for a given HTTP session. HTTP sessions are identified with a monitoring ID. By default, the monitoring ID for a given HTTP session is a random string (not the same as a session ID for security reasons). This monitoring ID can be configured by setting the `monitoring-attribute-name` element in session-descriptor of the `weblogic.xml` deployment descriptor and then setting a session attribute the defined `monitoring-attribute-name`. The `toString()` of the session attribute value will then be used as a monitoring ID. This element is useful for tagging session runtime information for different sessions. For example, you can set it to "username", if you have a "username" attribute that is unique.
`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.
`auth-cookie-id-length`	`20`	Defines the length of the secure cookie, `_WL_AUTHCOOKIE_JSESSIONID`. The default cookie length is 20, and the minimum cookie length is 8.

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 Java EE 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

`http://example.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>
```

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, the 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.

Table B-17 virtual-directory-mapping Elements

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.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

wl-dispatch-policy

Use the `wl-dispatch-policy` element to assign the Web application to a configured Work Manager by identifying the Work Manager name. This Web application-level parameter can be overridden by the dispatch policy setting at the individual servlet or JSP level. You can set the dispatch policy by using:

- The servlet's `wl-dispatch-policy`, using `<init-param>` of the `<servlet>` element in `web.xml`
- The `<dispatch-policy>` element in the `<servlet-descriptor>` element of `weblogic.xml`

> **Note**
>
> The `<dispatch-policy>` setting in `weblogic.xml` overrides the `wl-dispatch-policy` `<init-param>` configuration in `web.xml`.

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.

Table B-18 work-manager Elements

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) Cr)/R, where T is the average thread use time, R the arrival rate, and Cr 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 Cf/(P R T), where P is the percentage, R the arrival rate, T the average thread use time, and Cf 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table B-18 (Cont.) work-manager Elements

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.
`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.

Backward Compatibility Flags

For WebLogic Server, backward compatibility for WebLogic Server 9.2 or earlier is supported via the `backward-compatible` element within the `jsp-descriptor` element.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Compatibility with JSP 2.0 Web Applications

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

JSP 2.1 is supported as of WebLogic Server 10.0. Depending on the version of the Web application (version 2.4 or 2.5) and the setting of the `backward-compatible` element in the weblogic.xml descriptor file, WebLogic Server will also support JSP 2.0.

JSP Behavior and Buffer Suffix

- If a Web application version is 2.5 (for example, its `web.xml` has a version attribute of 2.5) *and* the `backward-compatible` flag is set to `false`, then:
  - All version 2.1 JSP/TAG files will follow the new JSP behavior.
  - All version 2.0 or earlier JSP/TAG files will follow the previous JSP 2.0 or earlier behavior.
- If a Web application version is 2.5 *and* the `backward-compatible` flag is set to `true`, then all JSP/TAG files will follow the previous JSP 2.0 or earlier behavior.
- If the Web application version is 2.4 or earlier, then all JSP/TAG files will follow the previous JSP 2.0 or earlier behavior no matter how the `backward-compatible` flag is set.

Implicit Servlet 2.5 Package Imports

The Servlet 2.5 specification mandates that only the `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, and `javax.servlet.http.*` packages be implicitly imported. In compliance with the Servlet 2.5 specification, WebLogic Server will only import these mandated packages. Whereas, previous releases of WebLogic Server also imported the `java.io.*`, `java.util.*`, and `javax.servlet.jsp.tagext.*` packages.

WebLogic Server will follow the previous 2.4 or earlier behavior and import the non-mandated packages, if any of the following occur:

- The `backward-compatible` flag is set to `true` in the weblogic.xml descriptor file.
- The Web application version is 2.4 or earlier.
- The individual JSP/TAG files in a version 2.5 Web application are version 2.0 or earlier.

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`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

C

Support for GlassFish Deployment Descriptors

Learn about WebLogic Server support for GlassFish deployment descriptors. WebLogic Server offers support for a subset of GlassFish deployment descriptors so that basic Web applications which deploy and run on GlassFish Server can be deployed on WebLogic Server.
If a Web application has both `weblogic.xml` and `glassfish-web.xml` or `sun-web.xml`, WebLogic Server will use `weblogic.xml` and ignore the GlassFish deployment descriptors. If a Web application has both `glassfish-web.xml` and `sun-web.xml`, WebLogic Server will use `glassfish-web.xml` and ignore `sun-web.xml`.

If the GlassFish element is on the list of supported deployment descriptors described in Table C-1, WebLogic Server will use the settings of its counterpart element in `weblogic.xml`. If the element is not on the list of supported deployment descriptors, WebLogic Server will ignore the element.

When `glassfish-web.xml` or `sun-web.xml` is being used, WebLogic Server emits an INFO level log message including whether individual settings are being used or ignored. WebLogic Server will not generate or persist the corresponding `weblogic.xml` descriptor elements.

> **Note**
>
> Web services do not support `glassfish-web.xml` deployment descriptor elements. If you are using Web services and define GlassFish elements in your Web application, the GlassFish deployment descriptors will not work.

Table C-1 Supported GlassFish Deployment Descriptors

glassfish-web.xml Element Name	Corresponding weblogic.xml Element Name
`context-root`	context-root
`security-role-mapping` • `role-name` • `principal-name` • `group-name`	security-role-assignment • `role-name` • `principal-name` • `principal-name`
`session-config` • `session-manager:manager-properties:reapIntervalSeconds` • `session-manager:manager-properties:maxSessions` • `session-manager:store-properties:directory` • `session-properties:timeoutSeconds`	session-descriptor • `invalidation-interval-seconds` • `max-in-memory-sessions` • `persistent-store-dir` • `timeout-secs`
`ejb-ref` • `ejb-ref-name` • `jndi-name`	ejb-reference-description • `ejb-ref-name` • `jndi-name`

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table C-1 (Cont.) Supported GlassFish Deployment Descriptors

glassfish-web.xml Element Name	Corresponding weblogic.xml Element Name
`resource-ref` • `res-ref-name` • `jndi-name`	resource-description • `res-ref-name` • `jndi-name`
`resource-env-ref` • `resource-env-ref-name` • `jndi-name`	resource-env-description • `resource-env-ref-name` • `jndi-name`
`class-loader` • `delegate`	container-descriptor • `prefer-web-inf-classes`
`jsp-config` • `checkInterval` • `keepgenerated` • `scratchdir`	jsp-descriptor • `page-check-seconds` • `keepgenerated` • `working-dir`

Copyright © 2007, 2026, Oracle and/or its affiliates.

D

Web Application Best Practices

Learn Oracle best practices for designing, developing, and deploying WebLogic Web applications and application resources in WebLogic Server.
This appendix includes the following sections:

CGI Best Practices

The following compatibility guidance describes how a CGI script invokes a subordinate script.

> **Note**
>
> **Compatibility guidance:** This section documents a WebLogic Server feature retained for existing applications. For new applications, use the Jakarta EE APIs and current-development approach identified in this section.

- 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

When writing HTTP servlets, review the recommended best practices.

- Compile your servlet classes into the `WEB-INF/classes` directory of your Web application.
- Register a servlet with the standard `@WebServlet` annotation or with a `servlet` and `servlet-mapping` entry in `web.xml`. The descriptor is optional when annotation-based registration is used.
- 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. 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

is needlessly time consuming in a production environment. If this attribute is set to `-1`, WebLogic Server does not check for modified servlets.

Best Practice When Subclassing ServletResponseWrapper

Jakarta EE provides the class `jakarta.servlet.ServletResponseWrapper`, which you can subclass in your Servlet to adapt its response.

Oracle recommends that if you create your own response wrapper by subclassing the `ServletResponseWrapper` class, you should always override the `flushBuffer()` and `resetBuffer()` methods. Not doing so might result in the response being committed prematurely.

Copyright © 2007, 2026, Oracle and/or its affiliates.

E

HTTP Proxy Servlet Parameters

Read descriptions of the `HttpProxyServlet` parameters.
This appendix includes the following topics:

Debug

Default: OFF

Sets the type of logging performed for debugging operations. The debugging information is written to `c:\TEMP\wlproxy.log` on Windows NT/2000 systems.

Override this location and filename by setting the `WLLogFile` parameter to a different directory and file. (See the `WLTempDir` parameter for an additional way to change this location.)

Ensure that the directory of the log file has write permission. Set any of the following logging options (HFC,HTW,HFW, and HTC options may be set in combination by entering them separated by commas, for example "HFC,HTW"):

- `ON`: The plug-in logs informational and error messages.
- `OFF`: No debugging information is logged.
- `HFC`: The plug-in logs headers from the client, informational, and error messages.
- `HTW`: The plug-in logs headers sent to WebLogic Server, and informational and error messages.
- `HFW`: The plug-in logs headers sent from WebLogic Server, and informational and error messages.
- `HTC`: The plug-in logs headers sent to the client, informational messages, and error messages.
- `ERR`: Prints only the Error messages in the plug-in.
- `ALL`: The plug-in logs headers sent to and from the client, headers sent to and from WebLogic Server, information messages, and error messages.

> **Note**
>
> For `HttpProxyServlet`, there are only two options: `TRUE(ON)` and `FALSE(OFF)`.

For information on setting logging without using the deprecated parameter, see Deprecated Directives for Apache HTTP Server.

DebugConfigInfo

Default: OFF

Enables the special query parameter "__WebLogicBridgeConfig". Use it to get details about configuration parameters from the proxy plug-in.

Copyright © 2007, 2026, Oracle and/or its affiliates.

For example, if you enable "__WebLogicBridgeConfig" by setting `DebugConfigInfo` and then send a request that includes the query string ?__WebLogicBridgeConfig, then the proxy plug-in gathers the configuration information and run-time statistics and returns the information to the browser. The proxy plug-in does not connect to Oracle WebLogic Server in this case.

This parameter is strictly for debugging and the format of the output message can change with releases. For security purposes, keep this parameter turned OFF in production systems.

DefaultFileName

Default: none

If the URI is "/" then the proxy plug-in performs the following steps:

1. Trims the path specified with the `PathTrim` parameter.
2. Appends the value of `DefaultFileName`.
3. Prepends the value specified with `PathPrepend`.

This procedure prevents redirects from Oracle WebLogic Server.

Set the DefaultFileName to the default welcome page of the Web application in Oracle WebLogic Server to which requests are being proxied. For example, If the `DefaultFileName` is set to welcome.html, an HTTP request like "http://somehost/weblogic" becomes "http://somehost/weblogic/welcome.html". For this parameter to function, the same file must be specified as a welcome file in all the Web Applications to which requests are directed. See Configuring Welcome Files in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Note for Apache users: If you are using Stronghold or Raven versions, define this parameter inside of a `Location` block, and not in an `IfModule` block.

FileCaching

Default: ON

When set to `ON`, and the size of the POST data in a request is greater than 2048 bytes, the POST data is first read into a temporary file on disk and then forwarded to Oracle WebLogic Server in chunks of 8192 bytes. This preserves the POST data during failover, allowing all necessary data to be repeated to the secondary if the primary goes down.

When `FileCaching` is `ON`, any client that tracks the progress of the POST will see that the transfer has completed even though the data is still being transferred between the WebServer and WebLogic. So, if you want the progress bar displayed by a browser during the upload to reflect when the data is actually available on the Oracle WebLogic Server, you might not want to have `FileCaching ON`.

When set to `OFF` and the size of the POST data in a request is greater than 2048 bytes, the reading of the POST data is postponed until an Oracle WebLogic Server cluster member is identified to serve the request. Then the proxy plug-in reads and immediately sends the POST data to Oracle WebLogic Server in chunks of 8192 bytes.

Turning `FileCaching OFF` limits failover. If the Oracle WebLogic Server primary server goes down while processing the request, the POST data already sent to the primary cannot be repeated to the secondary.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Finally, regardless of how `FileCaching` is set, if the size of the POST data is 2048 bytes or less the proxy plug-in will read the data into memory and use it if needed during failover to repeat to the secondary.

KeepAliveEnabled

Default: `ON`

This directive enables pooling of connections between the proxy plug-in and Oracle WebLogic Server. Valid values are `ON` and `OFF`.

While using Apache prefork mpm, Apache web server might fail. Set `KeepAliveEnabled` to `OFF` when using prefork mpm or use worker mpm in Apache.

> **Note**
>
> If both `KeepAliveEnabled` and HTTP/2 are configured for a back-end connection, the following message is generated:
> `KeepAliveEnabled option will be ignored since HTTP/2 connection is enabled`

KeepAliveSecs

Default: `20`

The length of time after which an inactive connection between the proxy plug-in and Oracle WebLogic Server is closed. You must set `KeepAliveEnabled` to `true` (`ON` when using the Apache HTTP Server) for this parameter to be effective.

The value of this parameter must be less than or equal to the value of the `Duration` field set in the Remote Console on the **Server > Protocols > HTTP** page, or the value set on the server MBean with the `KeepAliveSecs` attribute.

MaxPostSize

Default: 0

Maximum allowable size of POST data, in bytes. If the content-length exceeds `MaxPostSize`, the proxy plug-in returns an error message. If set to 0, the size of POST data is not checked. This is useful for preventing denial-of-service attacks that attempt to overload the server with POST data.

PathPrepend

Default: null

As per the RFC specification, generic syntax for URL is:

```
[PROTOCOL]://[HOSTNAME]:{PORT}/{PATH}/{FILENAME};{PATH_PARAMS}/{QUERY_STRING}...
```

`PathPrepend` specifies the path that the proxy plug-in prepends to the `{PATH}` portion of the original URL, after PathTrim is trimmed and before the request is forwarded to Oracle WebLogic Server.

Copyright © 2007, 2026, Oracle and/or its affiliates.

If you must append a File Name, use `DefaultFileName` parameter instead of `PathPrepend`.

PathTrim

Default: null

As per the RFC specification, generic syntax for URL is:

`[PROTOCOL]://[HOSTNAME]:{PORT}/{PATH}/{FILENAME};{PATH_PARAMS}/{QUERY_STRING}...`

`PathTrim` specifies the string trimmed by the proxy plug-in from the `{PATH}/{FILENAME}` portion of the original URL, before the request is forwarded to Oracle WebLogic Server. For example, if the `http://myWeb.server.com/weblogic/foo` URL is passed to the proxy plug-in for parsing and if `PathTrim` has been set to strip off `/weblogic`, before handing the URL to Oracle WebLogic Server, the URL forwarded to Oracle WebLogic Server is `http://myWeb.server.com:7001/foo`.

If you are newly converting an existing third-party server to proxy requests to Oracle WebLogic Server using the proxy plug-in, you will need to change application paths to `/foo` to include `weblogic/foo`. You can use `PathTrim` and `PathPrepend` in combination to change this path.

Configure the `PathTrim` parameter inside the `<Location>` tag.

The following configuration is **incorrect** because the `PathTrim` parameter is not configured inside the `<Location> tag`:

```
<Location /weblogic>
WLSRequest On
</Location>

<IfModule mod_weblogic.c>
WebLogicHost localhost
WebLogicPort 7001
PathTrim /weblogic
</IfModule>
```

The following configuration is **correct**:

```
<Location /weblogic>
 WLSRequest On
 PathTrim /weblogic
</Location>
```

The `<Location>` directive limits the scope of the enclosed directives by URL. See Apache Location Directive.

SecureProxy

Default: OFF

Set this parameter to ON to enable the use of the SSL protocol for all communication between the proxy plug-in and Oracle WebLogic Server. Remember to configure a port on the corresponding Oracle WebLogic Server for the SSL protocol before defining this parameter.

Copyright © 2007, 2026, Oracle and/or its affiliates.

This parameter may be set at two levels: in the configuration for the main server and—if you have defined any virtual hosts—in the configuration for the virtual host. The configuration for the virtual host inherits the SSL configuration from the configuration of the main server if the setting is not overridden in the configuration for the virtual host.

WebLogicHost

Required when proxying to a single Oracle WebLogic Server.

Default: none

Oracle WebLogic Server host (or virtual host name as defined in Oracle WebLogic Server) to which HTTP requests should be forwarded. If you are using a Oracle WebLogic cluster, use the `WebLogicCluster` parameter instead of `WebLogicHost`.

WebLogicPort

Required when proxying to a single Oracle WebLogic Server.

Default: none

The port at which Oracle WebLogic Server host is listening for connection requests from the proxy plug-in (or from other servers). (If you are using SSL between the proxy plug-in and Oracle WebLogic Server, set this parameter to the SSL listen port and set the `SecureProxy` parameter to `ON`).

If you are using a Oracle WebLogic Cluster, use the `WebLogicCluster` parameter instead of `WebLogicPort`.

WLCookieName

Default: `JSESSIONID`

If you change the name of the Oracle WebLogic Server session cookie in the Oracle WebLogic Server Web application, then you must change the `WLCookieName` parameter in the proxy plug-in to the same value. The name of the Oracle WebLogic session cookie is set in the WebLogic-specific deployment descriptor, in the `<session-descriptor>` element in `weblogic.xml`.

WLIOTimeoutSecs

New name for `HungServerRecoverSecs`.

Default: 120

Defines the amount of time the proxy plug-in waits for a response to a request from Oracle WebLogic Server. The proxy plug-in waits for `WLIOTimeoutSecs` for the server to respond, and then declares that the server is dead, and fails over to the next server. You must set the value to a large value. If the value is less than the time the servlets take to process, you might see unexpected results.

Minimum value: 10

Maximum value: 2147483647

Copyright © 2007, 2026, Oracle and/or its affiliates.

WLLogFile

Default: See the `Debug` parameter.

Specifies path and file name for the log file that is generated when the `Debug` parameter is set to `ON`. You must create this directory before setting this parameter.

For information on setting logging without using the deprecated parameter, see Deprecated Directives for Apache HTTP Server.

WLProxyPassThrough

Default: OFF

If you have a chained proxy setup, where a proxy plug-in is running behind some other proxy or load balancer, you must explicitly enable the `WLProxyPassThrough` parameter. This parameter allows the header to be passed through the chain of proxies.

WLProxySSL

Default: OFF

Set this parameter to `ON` to maintain SSL communication between the proxy plug-in and Oracle WebLogic Server when the following conditions exist:

- An HTTP client request specifies the HTTPS protocol.
- The request is passed through one or more proxy servers (including the Oracle WebLogic Server Proxy Plug-in).
- The connection between the proxy plug-in and Oracle WebLogic Server uses the HTTP protocol.

When `WLProxySSL` is set to `ON`, the location header returned to the client from Oracle WebLogic Server specifies the HTTPS protocol.

WLProxySSLPassThrough

Default: OFF

If a load balancer or other software deployed in front of the web server and proxy plug-in is the SSL termination point, and that product sets the `WL-Proxy-SSL` request header to `true` or `false` based on whether the client connected to it over SSL, set `WLProxySSLPassThrough` to `ON` so that the use of SSL is passed on to the Oracle WebLogic Server.

If the SSL termination point is in the web server where the proxy plug-in operates, or the load balancer does not set `WL-Proxy-SSL`, set `WLProxySSLPassThrough` to `OFF` (default).

Copyright © 2007, 2026, Oracle and/or its affiliates.