Oracle® Fusion Middleware

Developing Applications for Oracle WebLogic Server



26 (26.1.0.0.0)
G56908-02
November 2026



Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server, 26 (26.1.0.0.0)

G56908-02

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

Primary Author: Oracle Corporation

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.

Contents

Preface

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

1 Overview of WebLogic Server Application Development

WebLogic Server and the Jakarta EE Platform 1
Overview of Jakarta EE Applications and Modules 2
Web Application Modules 2
- Servlets 2
- Jakarta Server Pages 2
- More Information on Web Application Modules 3

Jakarta Enterprise Beans Modules 3
- EJB Documentation in WebLogic Server 3
- Additional EJB Information 3

Connector Modules 4
Enterprise Applications 4
- Jakarta EE Programming Model 4
- Packaging and Deployment Overview 4

WebLogic Web Services 5
JMS and JDBC Modules 6
WebLogic Diagnostic Framework Modules 6
- Using an External Diagnostics Descriptor 6
  - Defining an External Diagnostics Descriptor 7

Coherence Grid Archive (GAR) Modules 7
Bean Validation 7
XML Deployment Descriptors 8
- Automatically Generating Deployment Descriptors 13
- Java-Based Command-Line Utilities 13
- Upgrading Deployment Descriptors From Previous Releases of Jakarta EE and WebLogic Server 13

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

Deployment Plans 14
Development Tools 15
Java API Reference and the wls-api.jar File 15
Using the wls-api.jar File 15
Using the weblogic.jar File 16
Apache Ant 16
Using a Third-Party Version of Ant 17
Changing the Ant Heap Size 17
Source Code Editor or IDE 17
Database System and JDBC Driver 17
Web Browser 17
Third-Party Software 18

2 Using Ant Tasks to Configure and Use a WebLogic Server Domain

Overview of Configuring and Starting Domains Using Ant Tasks 1
Starting Servers and Creating Domains Using the wlserver Ant Task 1
Basic Steps for Using wlserver 2
Sample build.xml Files for wlserver 3
wlserver Ant Task Reference 3
Configuring a WebLogic Server Domain Using the wlconfig Ant Task 7
What the wlconfig Ant Task Does 7
Basic Steps for Using wlconfig 8
wlconfig Ant Task Reference 9
Main Attributes 9
Nested Elements 10
create 10
delete 11
set 11
get 12
query 12
invoke 13
Example of Creating a Security Realm with the wlconfig Ant Task 13
Using the libclasspath Ant Task 14
libclasspath Task Definition 14
libclasspath Ant Task Reference 14
Main libclasspath Attributes 14
Nested libclasspath Elements 15
librarydir 15
library 15
Example libclasspath Ant Task 15

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

3 Using the WebLogic Maven Plug-In

Installing Maven 1
Configuring the WebLogic Maven Plug-In 2
- How to use the WebLogic Maven Plug-in 2
- Basic Configuration POM File 5

Maven Plug-In Goals 6
- appc 7
- create-domain 11
- deploy 13
- distribute-app 17
- install 20
- list-apps 22
- purge-tasks 25
- redeploy 26
- remove-domain 29
- start-app 30
- start-server 32
- stop-app 34
- stop-server 36
- undeploy 38
- uninstall 41
- update-app 42
- wlst 45
- wlst-client 48
- ws-clientgen 54
- wsgen 58
- wsimport 62
- ws-wsdlc 69
- ws-jwsc 72

4 Creating a Split Development Directory Environment

Overview of the Split Development Directory Environment 1
- Source and Build Directories 1
- Deploying from a Split Development Directory 2
- Split Development Directory Ant Tasks 3

Using the Split Development Directory Structure: Main Steps 4
Organizing Jakarta EE Components in a Split Development Directory 4
- Source Directory Overview 5
- Enterprise Application Configuration 7
- Web Applications 7

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

EJBs 9
Important Notes Regarding EJB Descriptors 9
Organizing Shared Classes in a Split Development Directory 10
Shared Utility Classes 10
Third-Party Libraries 10
Class Loading for Shared Classes 11
Generating a Basic build.xml File Using weblogic.BuildXMLGen 11
weblogic.BuildXMLGen Syntax 12
Developing Multiple-EAR Projects Using the Split Development Directory 14
Organizing Libraries and Classes Shared by Multiple EARs 14
Linking Multiple build.xml Files 15
Best Practices for Developing WebLogic Server Applications 16

5 Building Applications in a Split Development Directory

Compiling Applications Using wlcompile 1
Using includes and excludes Properties 2
wlcompile Ant Task Attributes 2
Nested javac Options 2
Setting the Classpath for Compiling Code 2
Library Element for wlcompile and wlappc 3
Building Modules and Applications Using wlappc 3
wlappc Ant Task Attributes 4
wlappc Ant Task Syntax 5
Syntax Differences between appc and wlappc 6
weblogic.appc Reference 6
weblogic.appc Syntax 6
weblogic.appc Options 6

6 Deploying and Packaging from a Split Development Directory

Deploying Applications Using wldeploy 1
Packaging Applications Using wlpackage 1
Archive Versus Exploded Archive Directory 1
wlpackage Ant Task Example 2
wlpackage Ant Task Attribute Reference 2

7 Developing Applications for Production Redeployment

What is Production Redeployment? 1
Supported and Unsupported Application Types 1
Additional Application Support 2

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

Programming Requirements and Conventions 2
Applications Should Be Self-Contained 2
Versioned Applications Access the Current Version JNDI Tree by Default 3
Security Providers Must Be Compatible 3
Applications Must Specify a Version Identifier 3
Applications Can Access Name and Identifier 4
Client Applications Use Same Version when Possible 4
Assigning an Application Version 4
Application Version Conventions 4
Upgrading Applications to Use Production Redeployment 5
Accessing Version Information 5

8 Using Jakarta EE Annotations and Dependency Injection

Annotation Processing 1
Annotation Parsing 1
Deployment View of Annotation Configuration 1
Compiling Annotated Classes 2
Dynamic Annotation Updates 2
Dependency Injection of Resources 2
Application Life Cycle Annotation Methods 4
Standard Jakarta Annotations 4
jakarta.annotation.PostConstruct 4
jakarta.annotation.PreDestroy 5
jakarta.annotation.Resource 5
jakarta.annotation.Resources 7
Standard Jakarta Security Annotations 8
jakarta.annotation.security.DeclareRoles 8
jakarta.annotation.security.DenyAll 8
jakarta.annotation.security.PermitAll 9
jakarta.annotation.security.RolesAllowed 9
jakarta.annotation.security.RunAs 9

9 Using Contexts and Dependency Injection for the Jakarta EE Platform

About CDI for the Jakarta EE Platform 1
Defining a Managed Bean 2
Injecting a Bean 3
Defining the Scope of a Bean 3
Overriding the Scope of a Bean at the Point of Injection 4
Using Qualifiers 4
Defining Qualifiers for Implementations of a Bean Type 5

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

Applying Qualifiers to a Bean 6
Injecting a Qualified Bean 7
Providing Alternative Implementations of a Bean Type 8
Defining an Alternative Implementation of a Bean Type 8
Selecting an Alternative Implementation of a Bean Type for an Application 9
Applying a Scope and Qualifiers to a Session Bean 9
Applying a Scope to a Session Bean 10
Applying Qualifiers to a Session Bean 10
Using Producer Methods, Disposer Methods, and Producer Fields 10
Defining a Producer Method 10
Defining a Disposer Method 11
Defining a Producer Field 12
Initializing and Preparing for the Destruction of a Managed Bean 12
Initializing a Managed Bean 12
Preparing for the Destruction of a Managed Bean 13
Intercepting Method Invocations and Life Cycle Events of Bean Classes 13
Defining an Interceptor Binding Type 14
Defining an Interceptor Class 15
Identifying Methods for Interception 16
Enabling an Interceptor 17
Applying an Interceptor on a Producer 18
Decorating a Managed Bean Class 19
Defining a Decorator Class 19
Enabling a Decorator Class 21
Assigning an EL Name to a CDI Bean Class 21
Defining and Applying Stereotypes 22
Defining a Stereotype 23
Applying Stereotypes to a Bean 24
Using Events for Communications Between Beans 24
Defining an Event Type 24
Sending an Event 25
Handling an Event 26
Injecting a Predefined Bean 27
Injecting and Qualifying Resources 28
Using CDI With JCA Technology 30
Configuring a CDI Application 31
Enabling and Disabling CDI 32
Enabling and Disabling CDI for a Domain 32
Implicit Bean Discovery 33
Enabling and Disabling Implicit Bean Discovery for a Domain 34
Supporting Third-Party Portable Extensions 35
Using the Built-in Annotation Literals 36

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

Using the Configurator Interfaces 36
Bootstrapping a CDI Container 37

10 Duplicate - Jakarta JSON Processing

11 Jakarta JSON Processing

About JavaScript Object Notation (JSON) 1
Object Model API 1
Creating an Object Model from JSON Data 2
Creating an Object Model from Application Code 2
Navigating an Object Model 3
Writing an Object Model to a Stream 5
Streaming API 5
Reading JSON Data Using a Parser 6
Writing JSON Data Using a Generator 7
New Features for JSON Processing 8
JSON Pointer 8
JSON Patch 9
JSON Merge Patch 10

12 Jakarta JSON Binding

About Default Mapping 1
About Customized Mapping 2
JSON-B Support in Jakarta REST 3

13 Understanding WebLogic Server Application Classloading

Java Classloading 1
Java Classloader Hierarchy 1
Loading a Class 1
prefer-web-inf-classes Element 2
Changing Classes in a Running Program 2
Policy Class Loader and Historical Class Caching 2
Class Caching With Application Class Data Sharing 3
WebLogic Server Application Classloading 3
Overview of WebLogic Server Application Classloading 4
Application Classloader Hierarchy 4
Custom Module Classloader Hierarchies 5
Declaring the Classloader Hierarchy 6

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

User-Defined Classloader Restrictions 8
Servlet Reloading Disabled 8
Nesting Depth 8
Module Types 8
Duplicate Entries 8
Interfaces 9
Call-by-Value Semantics 9
In-Flight Work 9
Development-Only Reloading 9
Individual EJB Classloader for Implementation Classes 9
Application Classloading and Pass-by-Value or Reference 11
Using a Filtering Classloader 11
What is a Filtering Classloader 11
Configuring a Filtering Classloader 12
Resource Loading Order 12
Resolving Class References Between Modules and Applications 14
About Resource Adapter Classes 14
Packaging Shared Utility Classes 14
Manifest Class-Path 14
Using the Classloader Analysis Tool (CAT) 15
Opening the CAT Interface 15
How CAT Analyzes Classes 16
Identifying Class References through Manifest Hierarchies 16
Sharing Applications and Modules By Using Jakarta EE Libraries 17
Adding JARs to the Domain /lib Directory 17

14 Creating Shared Jakarta EE Libraries and Optional Packages

Overview of Shared Jakarta EE Libraries and Optional Packages 1
Library Directories 2
Versioning Support for Libraries 2
Shared Jakarta EE Libraries and Optional Packages Compared 3
Additional Information 3
Creating Shared Jakarta EE Libraries 4
Assembling Shared Jakarta EE Library Files 4
Assembling Optional Package Class Files 5
Editing Manifest Attributes for Shared Jakarta EE Libraries 5
Packaging Shared Jakarta EE Libraries for Distribution and Deployment 7
Referencing Shared Jakarta EE Libraries in an Enterprise Application 7
Overriding context-roots Within a Referenced Enterprise Library 9
URIs for Shared Jakarta EE Libraries Deployed As a Standalone Module 10
Referencing Optional Packages from a Jakarta EE Application or Module 10

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

Using weblogic.appmerge to Merge Libraries 12
Using weblogic.appmerge from the CLI 12
Using weblogic.appmerge as an Ant Task 13
Integrating Shared Jakarta EE Libraries with the Split Development Directory Environment 13
Deploying Shared Jakarta EE Libraries and Dependent Applications 13
Web Application Shared Jakarta EE Library Information 14
Using WebApp Libraries With Web Applications 14
Accessing Registered Shared Jakarta EE Library Information with LibraryRuntimeMBean 15
Order of Precedence of Modules When Referencing Shared Jakarta EE Libraries 15
Best Practices for Using Shared Jakarta EE Libraries 16

15 Programming Application Life Cycle Events

Understanding Application Life Cycle Events 1
Registering Events in weblogic-application.xml 2
Programming Basic Life Cycle Listener Functionality 2
Configuring a Role-Based Application Life Cycle Listener 3
Examples of Configuring Life Cycle Events with and without the URI Parameter 4
Understanding Application Life Cycle Event Behavior During Redeployment 5
Programming Application Version Life Cycle Events 5
Understanding Application Version Life Cycle Event Behavior 6
Types of Application Version Life Cycle Events 6
Example of Production Deployment Sequence When Using Application Version Life Cycle Events 7

16 Programming Context Propagation

Understanding Context Propagation 1
Programming Context Propagation: Main Steps 2
Programming Context Propagation in a Client 3
Programming Context Propagation in an Application 4

17 Programming Jakarta Mail with WebLogic Server

Overview of Using Jakarta Mail with WebLogic Server Applications 1
Understanding Jakarta Mail Configuration Files 1
Configuring Jakarta Mail for WebLogic Server 2
Sending Messages with Jakarta Mail 3
Reading Messages with Jakarta Mail 5

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

18 Threading and Clustering Topics

Using Threads in WebLogic Server 1
Thread Scheduling APIs 2
Programming Applications for WebLogic Server Clusters 3

19 Developing OSGi Bundles for WebLogic Server Applications

Understanding OSGi 1
Features Provided in WebLogic Server OSGi Implementation 1
Configuring the OSGi Framework 2
- Configuring OSGi Framework Instances 2
  - OSGi Framework Configuration in config.xml 3
  - Configuring OSGi Framework Instance From WLST 3
  - Configuring OSGi Framework Instance from a Java Program 5
  - Parameter Required for Installing Bundles in the Framework 7
- Configuring OSGi Framework Persistence 7
- Using OSGi Services 7
- Connecting to an OSGi Console 8

Creating OSGi Bundles 9
Deploying OSGi Bundles 9
- Preparing to Deploy an OSGi Bundle on a Target System 9
  - Preparing to Deploy Bundles as Enterprise Applications 9
  - Preparing to Deploy Bundles as Web Applications 10
  - Global Work Managers 10
  - Global Data Sources 11
- Deploying OSGi Bundles in the osgi-lib Directory 13
  - Setting Bundle and Framework Start Levels 13

Accessing Deployed Bundle Objects From JNDI 13
Using OSGi Logging Via WebLogic Server 15
Configuring a Filtering ClassLoader for OSGi Bundles 15
OSGI Example 16

20 Using the WebSocket Protocol in WebLogic Server

Understanding the WebSocket Protocol 1
- Limitations of the HTTP Request-Response Model 1
- WebSocket Endpoints 1
- Handshake Requests in the WebSocket Protocol 2
- Messaging and Data Transfer in the WebSocket Protocol 2

Understanding the WebLogic Server WebSocket Implementation 3
- WebSocket Protocol Implementation 3

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

WebLogic WebSocket Java API 3
Protocol Fallback for WebSocket Messaging 4
Sample WebSocket Applications 4
Overview of Creating a WebSocket Application 4
Creating an Endpoint 4
Creating an Annotated Endpoint 5
Creating a Programmatic Endpoint 6
Specifying the Path Within an Application to a Programmatic Endpoint 7
Handling Life Cycle Events for a WebSocket Connection 8
Handling Life Cycle Events in an Annotated WebSocket Endpoint 8
Handling a Connection Opened Event 9
Handling a Message Received Event 9
Handling an Error Event 11
Handling a Connection Closed Event 12
Handling Life Cycle Events in a Programmatic WebSocket Endpoint 12
Defining, Injecting, and Accessing a Resource for a WebSocket Endpoint 14
Sending a Message 15
Sending a Message to a Single Peer of an Endpoint 15
Sending a Message to All Peers of an Endpoint 16
Ensuring Thread Safety for WebSocket Endpoints 18
Encoding and Decoding a WebSocket Message 18
Encoding a Java Object as a WebSocket Message 18
Decoding a WebSocket Message as a Java Object 20
Specifying a Part of an Endpoint Deployment URI as an Application Parameter 22
Maintaining Client State 24
Configuring a Server Endpoint Programmatically 24
Building Applications that Use the Jakarta API for WebSocket 26
Deploying a WebSocket Application 27
Monitoring WebSocket Applications 28
Using WebSockets with Proxy Servers 30
Writing a WebSocket Client 31
Writing a Browser-Based WebSocket Client 31
Writing a Java WebSocket Client 32
Configuring a WebSocket Client Endpoint Programmatically 32
Connecting a Java WebSocket Client to a Server Endpoint 34
Setting the Maximum Number of Threads for Dispatching Messages from a WebSocket Client 35
Securing a WebSocket Application 36
Applying Verified-Origin Policies 36
Authenticating and Authorizing WebSocket Clients 37
Authorizing WebSocket Clients 38
Establishing Secure WebSocket Connections 38

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

Avoiding Mixed Content 39
Specifying Limits for a WebSocket Connection 39
Enabling Protocol Fallback for WebSocket Messaging 39
Using the JavaScript API for WebSocket Fallback in Client Applications 40
Configuring WebSocket Fallback 40
Creating a WebSocket Object 41
Handling Life Cycle Events for a JavaScript WebSocket Client 42
Sending a Message from a JavaScript WebSocket Client 44
Packaging and Specifying the Location of the WebSocket Fallback Client Library 44
Enabling WebSocket Fallback 45
Migrating an Application to the JSR 356 Java API for WebSocket from the Deprecated API 45
Comparison of the Jakarta WebSocket API and Proprietary WebLogic Server WebSocket API 45
Converting a Proprietary WebSocket Server Endpoint to Use the Jakarta WebSocket API 47
Replacing the /* Suffix in a Path Pattern String 49
Replacing a /* Suffix that Represents Variable Path Parameters in an Endpoint URI 50
Replacing a /* Suffix that Represents Additional Data for an Endpoint 50
Example of Converting a Proprietary WebSocket Server Endpoint to Use the Jakarta WebSocket API 51
Example of Using the Java API for WebSocket with WebLogic Server 52

A Enterprise Application Deployment Descriptor Elements

weblogic-application.xml Deployment Descriptor Elements A-1
weblogic-application A-1
ejb A-7
entity-cache A-8
max-cache-size A-9
xml A-10
parser-factory A-10
entity-mapping A-11
jdbc-connection-pool A-11
connection-factory A-12
pool-params A-13
driver-params A-17
security A-19
application-param A-20
classloader-structure A-20
listener A-20
singleton-service A-21
startup A-22
shutdown A-22

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

work-manager A-23
session-descriptor A-24
library-ref A-27
library-context-root-override A-28
fast-swap A-28
weblogic-application.xml Schema A-29
application.xml Schema A-29

B wldeploy Ant Task Reference

Overview of the wldeploy Ant Task B-1
Basic Steps for Using wldeploy B-1
Sample build.xml Files for wldeploy B-2
wldeploy Ant Task Attribute Reference B-4
Main Attributes B-4
Nested <files> Child Element B-8

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

Preface

This document describes building WebLogic Server e-commerce applications using the Jakarta EE Platform.

Audience

This document is written for application developers who want to build WebLogic Server applications using the Jakarta EE Platform. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

WebLogic Server applications are created by Java programmers, Web designers, and application assemblers. Programmers and designers create modules that implement the business and presentation logic for the application. Application assemblers assemble the modules into applications that are ready to deploy on WebLogic Server.

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 customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

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.

Related Documentation

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*.

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

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 text entered by users.
`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

Overview of WebLogic Server Application Development

Learn basic concepts about WebLogic Server applications, modules, and deployment descriptors.

WebLogic Server and the Jakarta EE Platform

WebLogic Server Jakarta EE applications are based on standardized, modular components. WebLogic Server provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming. Jakarta EE defines module behaviors and packaging in a generic, portable way, postponing runtime configuration until the module is deployed on an application server.

WebLogic Server implements Jakarta EE Platform 11 technologies (see https://jakarta.ee/specifications/platform/11/). Jakarta EE is the standard platform for developing multi-tier enterprise applications based on the Java programming language. The technologies that make up Jakarta EE were developed collaboratively by several software vendors.

Jakarta EE 11 Platform Highlights

Jakarta EE 11 continues to improve API and programming models needed for today's applications and adds features requested by the community. This release modernizes support for many industry standards and continues simplification of enterprise ready APIs. The key goals of the Jakarta EE 11 platform are to modernize the infrastructure for enterprise Java for the cloud and microservices environments, emphasize HTML5 and HTTP/2 support, enhance ease of development through new Contexts and Dependency Injection features, and further enhance security and reliability of the platform.

WebLogic Server and Jakarta EE Applications

WebLogic Server Jakarta EE applications are based on standardized, modular components. WebLogic Server provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming. Jakarta EE defines module behaviors and packaging in a generic, portable way, postponing run-time configuration until the module is actually deployed on an application server.

Jakarta EE includes deployment specifications for Web applications, EJB modules, Web services, enterprise applications, client applications, and connectors. Jakarta EE does not specify *how* an application is deployed on the target server—only how a standard module or application is packaged. For each module type, the specifications define the files required and their location in the directory structure.

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

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

Overview of Jakarta EE Applications and Modules

A WebLogic Server Jakarta EE application consists of one of the following modules or applications running on WebLogic Server: Web application modules, Jakarta Enterprise Beans (EJB) modules, connector modules, enterprise applications, or Web services.

- Web application modules—HTML pages, servlets, Jakarta Server Pages (JSP), and related files. See Web Application Modules.
- Jakarta Enterprise Beans (EJB) modules—entity beans, session beans, and message-driven beans. See Jakarta Enterprise Beans Modules.
- Connector modules—resource adapters. See Connector Modules.
- Enterprise applications—Web application modules, EJB modules, resource adapters and Web services packaged into an application. See Enterprise Applications.
- Web services—See WebLogic Web Services.

A WebLogic application can also include the following WebLogic-specific modules:

- JDBC and JMS modules—See JMS and JDBC Modules.
- WebLogic Diagnostic FrameWork (WLDF) modules—See WebLogic Diagnostic Framework Modules.
- Coherence Grid Archive (GAR) Modules—See Coherence Grid Archive (GAR) Modules.

Web Application Modules

A Web application on WebLogic Server can include the following files:

- Servlets, Jakarta Server Pages (JSP), and supporting Java classes, when the application needs server-side processing.
- Optionally, a `web.xml` deployment descriptor, a Jakarta EE standard XML document that describes the contents of a WAR file.
- Optionally, a `weblogic.xml` deployment descriptor, an XML document containing WebLogic Server-specific elements for Web applications.
- A Web application can also include HTML and XML pages with supporting files such as images and multimedia files.

Servlets

Servlets are Java classes that execute in WebLogic Server, accept a request from a client, process it, and optionally return a response to the client. An `HttpServlet` is most often used to generate dynamic Web pages in response to Web browser requests.

Jakarta Server Pages

Jakarta Server Pages (JSP) are Web pages coded with an extended HTML that makes it possible to embed Java code in a Web page. JSPs can call custom Java classes, known as tag libraries, using HTML-like tags. The `appc` compiler compiles JSPs and translates them into servlets. WebLogic Server automatically compiles JSPs if the servlet class file is not present or is older than the JSP source file. See Building Modules and Applications Using wlappc.

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

You can also precompile JSPs and package the servlet class in a Web application (WAR) file to avoid compiling in the server. Servlets and JSPs may require additional helper classes that must also be deployed with the Web application.

More Information on Web Application Modules

See the following documentation:

- Organizing Jakarta EE Components in a Split Development Directory.
- *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*
- *Developing JSP Tag Extensions for Oracle WebLogic Server*

Jakarta Enterprise Beans Modules

Jakarta Enterprise Beans (EJB) technology is the server-side component architecture for the development and deployment of component-based business applications. EJB technology enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Jakarta EE 11 technology.

The EJB 3.2 specification provides simplified programming and packaging model changes. The mandatory use of Java interfaces from previous versions has been removed, allowing plain old Java objects to be annotated and used as EJB components. The simplification is further enhanced through the ability to place EJB modules directly inside of Web applications, removing the need to produce archives to store the Web and EJB components and combine them together in an EAR file.

EJB Documentation in WebLogic Server

For more information about using EJBs with WebLogic Server, see:

- For information about all the new features in EJB, see New Features and Changes in EJB in *Developing Jakarta Enterprise Beans for Oracle WebLogic Server*.
- For information about basic EJB concepts and components, see Jakarta Enterprise Beans (EJBs) in *Understanding Oracle WebLogic Server*.
- For instructions on how to program, package, and deploy Jakarta Enterprise Beans on WebLogic Server, see Developing Jakarta Enterprise Beans for Oracle WebLogic Server.
- For instructions on how to organize and build WebLogic Server EJBs in a split directory environment, see Creating a Split Development Directory Environment.
- For more information on how to program and package EJBs, see *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Additional EJB Information

To learn more about EJB concepts, such as the benefits of enterprise beans, the types of enterprise beans, and their life cycles, visit the following Web sites:

- EJB 3.2 Specification (JSR-345) at `http://jcp.org/en/jsr/summary?id=345`
- The *Enterprise Beans* chapter of the Jakarta EE 11 Tutorial at `https://jakarta.ee/learn/docs/jakartaee-tutorial/11/entbeans/ejb-intro/ejb-intro.html`
- Jakarta EE 11 Platform: `https://jakarta.ee/specifications/platform/11/`

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

Connector Modules

Connectors (also known as resource adapters) contain the Java, and if necessary, the native modules required to interact with an Enterprise Information System (EIS). A resource adapter deployed to the WebLogic Server environment enables Jakarta EE applications to access a remote EIS. WebLogic Server application developers can use HTTP servlets, Jakarta Server Pages (JSP), Jakarta Enterprise Beans (EJB), and other APIs to develop integrated applications that use the EIS data and business logic.

To deploy a resource adapter to WebLogic Server, you must first create and configure WebLogic Server-specific deployment descriptor, `weblogic-ra.xml` file, and add this to the deployment directory. Resource adapters can be deployed to WebLogic Server as standalone modules or as part of an enterprise application. See Enterprise Applications.

For more information on connectors, see *Developing Resource Adapters for Oracle WebLogic Server*.

Enterprise Applications

An enterprise application consists of one or more Web application modules, EJB modules, and resource adapters. It might also include a client application.

An enterprise application can be optionally defined by an `application.xml` file, which was the standard Jakarta EE deployment descriptor for enterprise applications.

Jakarta EE Programming Model

An important aspect of the Jakarta EE programming model is the introduction of metadata annotations. Annotations simplify the application development process by allowing a developer to specify within the Java class itself how the application 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 (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 annotations to declare component metadata and request container services, including dependency injection in Web and enterprise-bean components. See the Jakarta EE 11 API documentation. See Using Jakarta EE Annotations and Dependency Injection.

If the application includes WebLogic Server-specific extensions, the application is further defined by a weblogic-application.xml file. An application-client module uses META-INF/application-client.xml for its standard deployment descriptor and may include META-INF/weblogic-application-client.xml for WebLogic-specific configuration. See Enterprise Application Deployment Descriptor Elements.

Packaging and Deployment Overview

For both production and development purposes, Oracle recommends that you package and deploy even standalone Web applications, EJBs, and resource adapters as part of an enterprise application. Doing so allows you to take advantage of Oracle's split development directory structure, which greatly facilitates application development. See Creating a Split Development Directory Environment.

An enterprise application consists of Web application modules, EJB modules, and resource adapters. It can be packaged as follows:

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

- For development purposes, Oracle recommends the WebLogic split development directory structure. Rather than having a single archived EAR file or an exploded EAR directory structure, the split development directory has two parallel directories that separate source files and output files. This directory structure is optimized for development on a single WebLogic Server instance. See Creating a Split Development Directory Environment. Oracle provides the `wlpackage` Ant task, which allows you to create an EAR without having to use the JAR utility; this is exclusively for the split development directory structure. See Packaging Applications Using wlpackage.
- For development purposes, Oracle further recommends that you package standalone Web applications and Jakarta Enterprise Beans (EJB) as part of an enterprise application, so that you can take advantage of the split development directory structure. See Organizing Jakarta EE Components in a Split Development Directory.
- For production purposes, Oracle recommends the exploded (unarchived) directory format. This format enables you to update files without having to redeploy the application. To update an archived file, you must unarchive the file, update it, then rearchive and redeploy it.
- You can choose to package your application as a JAR archived file using the `jar` utility with an `.ear` extension. Archived files are easier to distribute and take up less space. An EAR file contains all of the JAR, WAR, and RAR module archive files for an application and an XML descriptor that describes the bundled modules. See Packaging Applications Using wlpackage.

The optional `META-INF/application.xml` deployment descriptor contains an element for each Web application, EJB, and connector module, as well as additional elements to describe security roles and application resources such as databases. If this descriptor is present the WebLogic deployer picks the list of modules from this descriptor. However if this descriptor is not present, the container guesses the modules from the annotations defined on the POJO (plain-old-Java-object) classes. See Enterprise Application Deployment Descriptor Elements.

WebLogic Web Services

Web services can be shared by and used as modules of distributed Web-based applications. They commonly interface with existing back-end applications, such as customer relationship management systems, order-processing systems, and so on. Web services can reside on different computers and can be implemented by vastly different technologies, but they are packaged and transported using standard Web protocols, such as HTTP, thus making them easily accessible by any user on the Web.

A Web service consists of the following modules, at a minimum:

- A Web service implementation hosted by a server on the Web. WebLogic JAX-WS web services are hosted by WebLogic Server. A Web service module may include either Java classes or EJBs that implement the Web service. Web services are packaged either as Web application archives (WARs) or EJB modules (JARs), depending on the implementation.
- A standard for transmitting data and Web service invocation calls between the Web service and the user of the Web service. WebLogic JAX-WS web services use Simple Object Access Protocol (SOAP) 1.1 as the message format and HTTP as the connection protocol.
- A standard for describing the Web service to clients so they can invoke it. WebLogic Web services use Web services Description Language (WSDL) 1.1, an XML-based specification, to describe themselves.
- A standard for clients to invoke Web services—JAX-WS. See *Developing JAX-WS Web Services for Oracle WebLogic Server*.

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

WebLogic Server also includes support for RESTful web services. For more information about RESTful web services, see *Developing and Securing RESTful Web Services for Oracle WebLogic Server*.

For more information about WebLogic Web services and the standards that are supported, see *Understanding WebLogic Web Services for Oracle WebLogic Server*.

JMS and JDBC Modules

JMS and JDBC configurations are stored as modules, defined by an XML file that conforms to the `weblogic-jms.xsd` and `jdbc-data-source.xsd` schema, respectively. These modules are similar to standard Jakarta EE modules. An administrator can create and manage JMS and JDBC modules as global system resources, as modules packaged with a Jakarta EE application (as a packaged resource), or as standalone modules that can be made globally available.

With modular deployment of JMS and JDBC resources, you can migrate your application and the required JMS or JDBC configuration from environment to environment, such as from a testing environment to a production environment, without opening an enterprise application file (such as an EAR file) or a JMS or JDBC standalone module, and without extensive manual JMS or JDBC reconfiguration.

Application developers create application modules in an enterprise-level IDE or another development tool that supports editing of XML files, then package the JMS or JDBC modules with an application and pass the application to a WebLogic administrator to deploy.

For more information, see:

- Configuring JMS Application Modules for Deployment
- Configuring JDBC Application Modules for Deployment

WebLogic Diagnostic Framework Modules

The WebLogic Diagnostic Framework (WLDF) provides features for generating, gathering, analyzing, and persisting diagnostic data from WebLogic Server instances and from applications deployed to server instances.

For server-scoped diagnostics, some WLDF features are configured as part of the configuration for the domain. Other features are configured as system resource descriptors that can be targeted to servers (or clusters). For application-scoped diagnostics, diagnostic features are configured as resource descriptors for the application.

Application-scoped instrumentation is configured and deployed as a diagnostic module, which is similar to a diagnostic system module. However, an application module is configured in an XML configuration file named `weblogic-diagnostics.xml` which is packaged with the application archive.

For detailed instructions for configuring instrumentation for applications, see Configuring Application-Scoped Instrumentation.

Using an External Diagnostics Descriptor

WebLogic Server also supports the use of an external diagnostics descriptor so you can integrate diagnostic functionality into an application that has not imported diagnostic descriptors. This feature supports the deployment view and deployment of an application or a

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

module, detecting the presence of an external diagnostics descriptor if the descriptor is defined in your deployment plan (`plan.xml`).

Defining an External Diagnostics Descriptor

First, define the diagnostic descriptor as external and configure its URI in the `plan.xml` file. For example:

```
<deployment-plan xmlns="http://xmlns.oracle.com/weblogic/deployment-plan">
  <application-name>reviewService</application-name>
  <module-override>
    <module-name>reviewService.ear</module-name>
    <module-type>ear</module-type>
    <module-descriptor external="true">
      <root-element>wldf-resource</root-element>
      <uri>META-INF/weblogic-diagnostics.xml</uri>
    </module-descriptor>
  </module-override>
  <config-root>D:\plan</config-root>
</deployment-plan>
```

Save this plan as plan.xml and place the external diagnostics descriptor at D:\plan\META-INF\weblogic-diagnostics.xml. The module-name must match the root application filename.

Coherence Grid Archive (GAR) Modules

A Coherence GAR module provides distributed in-memory caching and data grid computing that allows applications to increase their availability, scalability, and performance. GAR modules are deployed as both standalone modules and packaged with Jakarta EE applications (as a packaged resource). A GAR module may also be made globally available.

A GAR module is defined by the coherence-application.xml deployment descriptor and must conform to the `coherence-application.xsd` XML schema. The GAR contains the artifacts that comprise a Coherence application: Coherence configuration files, application classes (such as entry processors, aggregators, filters), and any dependencies that are required.

Bean Validation

The Bean Validation specification (JSR 380) defines a metadata model and API for validating data in JavaBeans components. It is supported on both the server and Jakarta EE 11 client; therefore, instead of distributing validation of data over several layers, such as the browser and the server side, you can define the validation constraints in one place and share them across the different layers.

Bean validation is not only for validating beans. In fact, it can also be used to validate any Java object.

Bean Validation and JNDI

Where required by the Jakarta EE specifications, the default `Validator` and `ValidatorFactory` are located using JNDI under the names `java:comp/Validator` and `java:comp/ValidatorFactory`. These two artifacts reflect the validation descriptor that is in scope.

Bean Validation Configuration

Bean validation can be configured by using XML descriptors or annotation.

- Descriptors:

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

  - Descriptor elements override corresponding annotations.
  - Weblogic Server allows one descriptor per module. Therefore, an application can have several validation descriptors but only one is allowed per module scope.
  - Validation descriptors are named `validation.xml` and are packaged in the `META-INF` directory, except for Web modules, where the descriptor is packaged in the `WEB-INF` directory.
- Annotations:
  - Injection of the default `Validator` and `ValidatorFactory` is requested using the `@Resource` annotation. However, not all source files are scanned for this annotation.
  - The WebLogic Connector uses bean validation internally to validate the connector descriptors.

Once bean validation is configured, the standard set of container managed classes for a given container will be scanned. For example, for EJBs, bean and interceptor classes are scanned. Web application classes and ManagedBeans also support the injection of `Validator` and `ValidatorFactories`.

For more information about the classes that support bean validation, please see the related component specifications for the list of classes that support dependency injection.

XML Deployment Descriptors

A *deployment configuration* refers to the process of defining the deployment descriptor values required to deploy an enterprise application to a particular WebLogic Server domain. The deployment configuration for an application or module is stored in three types of XML document: Jakarta EE deployment descriptors, WebLogic Server descriptors, and WebLogic Server deployment plans.

This section describes the Jakarta EE and WebLogic-specific deployment descriptors. See Deployment Plans for information on deployment plans.

The Jakarta EE programming model uses the JDK annotations feature for Web containers, such as EJBs, servlets, Web applications, and JSPs. Annotations simplify the application development process by allowing a developer to specify within the Java class itself how the 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 Web applications (2.4 and earlier), enterprise applications (1.4 and earlier), and EJBs (2.*x* and earlier). See Using Jakarta EE Annotations and Dependency Injection.

However, enterprise applications fully support the use of deployment descriptors, even though the standard Jakarta EE ones are not required. For example, you may prefer to use the old EJB 2.*x* programming model, or might want to allow further customizing of the EJB at a later development or deployment stage; in these cases you can create the standard deployment descriptors in addition to, or instead of, the metadata annotations.

Modules and applications have deployment descriptors—XML documents—that describe the contents of the directory or JAR file. Deployment descriptors are text documents formatted with XML tags. The Jakarta EE specifications define standard, portable deployment descriptors for Jakarta EE modules and applications. Oracle defines additional WebLogic-specific deployment descriptors for deploying a module or application in the WebLogic Server environment.

Table 1-1 lists the types of modules and applications and their Jakarta EE-standard and WebLogic-specific deployment descriptors.

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

> **Note**
>
> The XML schemas for the WebLogic deployment descriptors listed in the following table include elements from the http://xmlns.oracle.com/weblogic/weblogic-jakartaee/2.0/weblogic-jakartaee.xsd schema, which describes common elements shared among all WebLogic-specific deployment descriptors.
>
> For the most current schema information, see https://www.oracle.com/webfolder/technetwork/weblogic/wls_15.1.1.0.0.html

Table 1-1 Jakarta EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Web Application	Jakarta EE	`web.xml` See the Servlet 5.0 Schema at https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd `WEB-INF/beans.xml`—required only if the classes in the WAR file are to participate in Contexts and Dependency Injection (CDI) Schema: https://jakarta.ee/xml/ns/jakartaee/beans_3_0.xsd See Using Contexts and Dependency Injection for the Jakarta EE Platform.
Web Application	WebLogic	`weblogic.xml` Schema: http://xmlns.oracle.com/weblogic/weblogic-web-app/2.0/weblogic-web-app.xsd See weblogic.xml Deployment Descriptor Elements in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.
Enterprise Bean 4.0	Jakarta EE 11	`ejb-jar.xml` See the EJB 4.0 Schema at `http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/ejb-jar_3_2.xsd` `META-INF/beans.xml`—required only if the classes in the EJB JAR file are to participate in CDI Schema: `http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/beans_2_0.xsd` See Using Contexts and Dependency Injection for the Jakarta EE Platform.

Enterprise Bean 3.2	WebLogic	`weblogic-ejb-jar.xml`
		See the EJB 3.2 Schema at: http://xmlns.oracle.com/weblogic/weblogic-ejb-jar/2.0/weblogic-ejb-jar.xsd`weblogic-rdbms-jar.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-rdbms-jar/2.0/weblogic-rdbms-jar.xsd
		`persistence-configuration.xml`
		Schema: `http://xmlns.oracle.com/weblogic/persistence-configuration/1.0/persistence-configuration.xsd`
		See *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

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

Table 1-1 (Cont.) Jakarta EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Enterprise Bean 3.0	Jakarta EE	`ejb-jar.xml` See the EJB 4.0 Schema at http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/ejb-jar_3_1.xsd `META-INF/beans.xml`—required only if the classes in the EJB JAR file are to participate in CDI Schema: http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/beans_1_1.xsd See Using Contexts and Dependency Injection for the Jakarta EE Platform.

Enterprise Bean 3.0	WebLogic	`weblogic-ejb-jar.xml`
		Schema http://xmlns.oracle.com/weblogic/weblogic-ejb-jar/1.6/weblogic-ejb-jar.xsd
		`weblogic-rdbms-jar.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-rdbms-jar/1.2/weblogic-rdbms-jar.xsd
		`persistence-configuration.xml`
		Schema: http://xmlns.oracle.com/weblogic/persistence-configuration/1.0/persistence-configuration.xsd
		See *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

Enterprise Bean 2.1	Jakarta EE	`ejb-jar.xml` See the EJB 2.1 Schema at http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd

Enterprise Bean 2.1	WebLogic	`weblogic-ejb-jar.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-ejb-jar/1.6/weblogic-ejb-jar.xsd
		See The weblogic-ejb-jar.xml Deployment Descriptor in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.
		`weblogic-cmp-rdbms-jar.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-rdbms-jar/1.2/weblogic-rdbms-jar.xsd
		See The weblogic-cmp-rdbms-jar.xml Deployment Descriptor in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

Web services	Jakarta EE	`webservices.xml` See the Web services 1.4 Schema at http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/javaee_web_services_1_4.xsd

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

Table 1-1 (Cont.) Jakarta EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors

Web services	WebLogic	`weblogic-webservices.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-webservices/1.1/weblogic-webservices.xsd
		`weblogic-wsee-clientHandlerChain.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-wsee-clientHandlerChain/1.0/weblogic-wsee-clientHandlerChain.xsd
		`weblogic-webservices-policy.xml`
		Schema: http://xmlns.oracle.com/weblogic/webservice-policy-ref/1.1/webservice-policy-ref.xsd
		`weblogic-wsee-standaloneclient.xml`
		Schema: http://xmlns.oracle.com/weblogic/weblogic-wsee-standaloneclient/1.0/weblogic-wsee-standaloneclient.xsd
		See WebLogic Web Service Deployment Descriptor Element Reference in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Resource Adapter	Jakarta EE	`ra.xml` See the Connector 1.7 Schema at http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/connector_1_7.xsd `META-INF/beans.xml`—required only if the classes in the RAR file are to participate in CDI Schema: http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/beans_2_0.xsd See Using Contexts and Dependency Injection for the Jakarta EE Platform.
Resource Adapter	WebLogic	`weblogic-ra.xml` Schema: http://xmlns.oracle.com/weblogic/weblogic-connector/1.5/weblogic-connector.xsd See weblogic-ra.xml Schema in *Developing Resource Adapters for Oracle WebLogic Server*.
Enterprise Application	Jakarta EE	`application.xml` See the Application 8 Schema at http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/application_8.xsd
Enterprise Application	WebLogic	`weblogic-application.xml` Schema: http://xmlns.oracle.com/weblogic/weblogic-application/1.8/weblogic-application.xsd See weblogic-application.xml Deployment Descriptor Elements.

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

Table 1-1 (Cont.) Jakarta EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Client Application	Jakarta EE	`application-client.xml` See the Application Client 8 Schema at http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/application-client_8.xsd `META-INF/beans.xml`—required only if the classes in the application client JAR file are to participate in CDI Schema: http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/beans_2_0.xsd See Using Contexts and Dependency Injection for the Jakarta EE Platform.
Client Application	WebLogic	`application-client.xml` Schema: http://xmlns.oracle.com/weblogic/weblogic-application-client/1.6/weblogic-application-client.xsd See Developing a Jakarta EE Application Client (Thin Client) in *Developing Stand-alone Clients for Oracle WebLogic Server*.
HTTP Pub/Sub Application	WebLogic	`weblogic-pubsub.xml` Schema: http://xmlns.oracle.com/weblogic/weblogic-pubsub/1.0/weblogic-pubsub.xsd See Using the HTTP Publish-Subscribe Server in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.
JMS Module	WebLogic	*FileName*`-jms.xml`, where *FileName* can be anything you want. Schema: http://xmlns.oracle.com/weblogic/weblogic-jms/1.8/weblogic-jms.xsd See Configuring JMS Application Modules for Deployment in *Administering JMS Resources for Oracle WebLogic Server*.
JDBC Module	WebLogic	*FileName*`-jdbc.xml`, where *FileName* can be anything you want. Schema: http://xmlns.oracle.com/weblogic/jdbc-data-source/1.6/jdbc-data-source.xsd See Configuring JDBC Application Modules for Deployment in *Administering JDBC Data Sources for Oracle WebLogic Server*.
Deployment Plan	WebLogic	`plan.xml` Schema: http://www.oracle.com/webfolder/technetwork/weblogic/deployment-plan/index.html See Understanding WebLogic Server Deployment in *Deploying Applications to Oracle WebLogic Server*.
Resource Deployment Plan	WebLogic	`resource-deployment-plan.xml` Schema: http://xmlns.oracle.com/weblogic/resource-deployment-plan/1.0/resource-deployment-plan.xsd
WLDF Module	WebLogic	`weblogic-diagnostics.xml` Schema: http://xmlns.oracle.com/weblogic/weblogic-diagnostics/2.0/weblogic-diagnostics.xsd See Deploying WLDF Application Modules in *Configuring and Using the Diagnostics Framework for Oracle WebLogic Server*.

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

Table 1-1 (Cont.) Jakarta EE and WebLogic Deployment Descriptors

Module or Application	Scope	Deployment Descriptors
Coherence Modules	WebLogic	`coherence-application.xml` Schema: http://xmlns.oracle.com/coherence/coherence-application/1.0/coherence-application.xsd See *Developing Oracle Coherence Applications for Oracle WebLogic Server*.

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

Automatically Generating Deployment Descriptors

WebLogic Server provides a variety of tools for automatically generating deployment descriptors. These are discussed in the sections that follow.

Java-Based Command-Line Utilities

WebLogic Server includes a set of Java-based command-line utilities that automatically generate both standard Jakarta EE and WebLogic-specific deployment descriptors for Web applications and enterprise applications.

These command-line utilities examine the classes you have assembled in a staging directory and build the appropriate deployment descriptors based on the servlet classes, and so on. These utilities include:

- `java weblogic.marathon.ddinit.EarInit` — automatically generates the deployment descriptors for enterprise applications.
- `java weblogic.marathon.ddinit.WebInit` — automatically generates the deployment descriptors for Web applications.

For an example of `DDInit`, assume that you have created a directory called `c:\stage` that contains the JSP files and other objects that make up a Web application but you have not yet created the `web.xml` and `weblogic.xml` deployment descriptors. To automatically generate them, execute the following command:

```
prompt> java weblogic.marathon.ddinit.WebInit c:\stage
```

The utility generates the `web.xml` and `weblogic.xml` deployment descriptors and places them in the `WEB-INF` directory, which `DDInit` will create if it does not already exist.

Upgrading Deployment Descriptors From Previous Releases of Jakarta EE and WebLogic Server

So that your applications can take advantage of the features in the current Jakarta EE specification and release of WebLogic Server, Oracle recommends that you always upgrade deployment descriptors when you migrate applications to a new release of WebLogic Server.

To upgrade the deployment descriptors in your Jakarta EE applications and modules, first use the `weblogic.DDConverter` tool to generate the upgraded descriptors into a temporary directory. Once you have inspected the upgraded deployment descriptors to ensure that they

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

are correct, repackage your Jakarta EE module archive or exploded directory with the new deployment descriptor files.

Invoke `weblogic.DDConverter` with the following command:

```
java weblogic.DDConverter -d output_directory [options] archive_file_or_directory
```

where *`archive_file_or_directory`* refers to the archive file (EAR, WAR, JAR, or RAR) or exploded directory of your enterprise application, Web application, EJB, or resource adapter.

The following table describes the `weblogic.DDConverter` command options.

Table 1-2 weblogic.DDConverter Command Options

Option	Description
`-d <dir>`	Required. Specifies the directory to which the upgraded descriptors are written.
`-help`	Prints the standard usage message.
`-quiet`	Turns off output messages except error messages.
`-verbose`	Turns on additional output used for debugging.

The following example shows how to use the `weblogic.DDConverter` command to generate upgraded deployment descriptors for the `my.ear` enterprise application into the subdirectory `tempdir` in the current directory:

```
prompt> java weblogic.DDConverter -d tempdir my.ear
```

Deployment Plans

A *deployment plan* is an XML document that defines an application's WebLogic Server deployment configuration for a specific WebLogic Server environment. A deployment plan resides outside of an application's archive file, and can apply changes to deployment properties stored in the application's existing WebLogic Server deployment descriptors.

Administrators use deployment plans to easily change an application's WebLogic Server configuration for a specific environment *without* modifying existing Jakarta EE or WebLogic-specific deployment descriptors. Multiple deployment plans can be used to reconfigure a single application for deployment to multiple, differing WebLogic Server environments.

After programmers have finished programming an application, they export its deployment configuration to create a custom deployment plan that administrators later use for deploying the application into new WebLogic Server environments. Programmers distribute both the application deployment files and the custom deployment plan to deployers (for example, testing, staging, or production administrators) who use the deployment plan as a blueprint for configuring the application for their environment.

WebLogic Server provides the following tools to help programmers export an application's deployment configuration:

- `weblogic.PlanGenerator` creates a template deployment plan with null variables for selected categories of WebLogic Server deployment descriptors. This tool is recommended if you are beginning the export process and you want to create a template deployment plan with null variables for an entire class of deployment descriptors.
- The WebLogic Remote Console updates or creates new deployment plans as necessary when you change configuration properties for an installed application. You can use the WebLogic Remote Console to generate a new deployment plan or to add or override

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

variables in an existing plan. The WebLogic Remote Console provides greater flexibility than `weblogic.PlanGenerator`, because it allows you to interactively add or edit individual deployment descriptor properties in the plan, rather than export entire categories of descriptor properties.

For complete and detailed information about creating and using deployment plans, see:

- Understanding WebLogic Server Deployment
- Exporting an Application for Deployment to New Environments
- Understanding WebLogic Server Deployment Plans

Development Tools

To develop WebLogic Server applications, you need various tools such as Java API Reference and the `wls-api.jar` file, source code editor or IDE, database system and JDBC driver, and Web browser. You also need third party tools such as Apache Ant.

This section describes required and optional tools for developing WebLogic Server applications.

Java API Reference and the wls-api.jar File

Oracle provides the Oracle Fusion Middleware Java API Reference for Oracle WebLogic Server, which defines all of the supported Java classes available for use when developing Jakarta EE applications for WebLogic Server. See the *Java API Reference for Oracle WebLogic Server*.

In conjunction with the Java API Reference for Oracle WebLogic Server, Oracle recommends using the `wls-api.jar` file to develop and compile Jakarta EE applications for your WebLogic Server environment. The `wls-api.jar` file is located in the `wlserver/server/lib` directory of your WebLogic Server distribution and offers the following benefits:

- Developing more performant code based on tested best practices
- Avoiding deprecated or unsupported code paths

See the following sections:

Using the wls-api.jar File

Use the `wls-api.jar` file and the `api.jar` file to develop and compile your Jakarta EE applications in Integrated Development Environments (IDEs), such as Oracle JDeveloper. IDEs provide an array of tools to simplify development of Java-based applications. The `wls-api.jar` file provides a clean and concise API jar to develop and run Jakarta EE applications for WebLogic environments.

> **Note**
>
> The `wls-api.jar` file does not reference any Jakarta EE classes. Oracle provides the `api.jar` file with a manifest classpath that includes access to Jakarta EE JARs.

You may need to include the `weblogic.jar` file in the classpath of your development environment to access tools such as WLST, the `weblogic.Deployer` utility, and `weblogic.appc`.

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

Using the weblogic.jar File

You must continue to use the `weblogic.jar` file for runtime environments, as a client or to develop and compile legacy applications. However, use the `wls-api.jar` file to develop and compile Jakarta EE applications for your WebLogic Server environment.

Apache Ant

The preferred Oracle method for building applications with WebLogic Server is Apache Ant. Ant is a Java-based build tool. One of the benefits of Ant is that is it is extended with Java classes, rather than shell-based commands. Oracle provides numerous Ant extension classes to help you compile, build, deploy, and package applications using the WebLogic Server split development directory environment.

Another benefit is that Ant is a cross-platform tool. Developers write Ant build scripts in eXtensible Markup Language (XML). XML tags define the targets to build, dependencies among targets, and tasks to execute in order to build the targets. Ant libraries are bundled with WebLogic Server to make it easier for our customers to build Java applications out of the box.

Before running WebLogic Ant tasks, set the WebLogic environment in the same command shell. On Windows, run ORACLE_HOME\wlserver\server\bin\setWLSEnv.cmd. In a Bourne-compatible UNIX shell, source the script as follows:

```
. "$ORACLE_HOME/wlserver/server/bin/setWLSEnv.sh"
ant -version
```

When building the installed samples, also follow the sample-specific environment instructions.

For a complete explanation of ant capabilities, see: https://ant.apache.org/manual/

> **Note**
>
> The Apache Ant website publishes online documentation for only the most current version of Ant, which might be different from the version of Ant that is bundled with WebLogic Server. Use the following command, after setting your WebLogic environment, to determine the version of Ant bundled with WebLogic Server:
>
> ```
> prompt> ant -version
> ```
>
> To view the documentation for a specific version of Ant, such as the version included with WebLogic Server, download the Ant zip file from https://archive.apache.org/dist/ant/binaries/ and extract the documentation.

For more information on using Ant to compile your cross-platform scripts or using cross-platform scripts to create XML scripts that can be processed by Ant, refer to any of the WebLogic Server examples, such as *ORACLE_HOME*`/wlserver/samples/server/examples/src/examples/ejb20/basic/beanManaged/build.xml`, 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*.

Also refer to the following WebLogic Server documentation on building examples using Ant: *ORACLE_HOME*`/wlserver/samples/server/examples/src/examples/examples.html`.

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

Using a Third-Party Version of Ant

You can use your own version of Ant if the one bundled with WebLogic Server is not adequate for your purposes. To determine the version of Ant that is bundled with WebLogic Server, run the following command after setting your WebLogic environment:

```
prompt> ant -version
```

Install the complete Ant distribution in a separate directory. Set ANT_HOME to that directory and place its bin directory first in PATH, after setting the WebLogic environment. Run ant -version to verify the selected version. Do not replace individual Ant JARs inside the WebLogic installation.

Changing the Ant Heap Size

Set ANT_OPTS before invoking Ant to pass heap options to the JVM that runs Ant. For example, set a maximum heap of 512 MB using the command for your shell:

```
# Bourne-compatible shells
export ANT_OPTS="-Xmx512m"

# csh or tcsh
setenv ANT_OPTS "-Xmx512m"


rem Windows Command Prompt
set "ANT_OPTS=-Xmx512m"
```

These examples set ANT_OPTS; preserve any other JVM options required by your build. For a persistent setting, configure ANT_OPTS in your user environment or project wrapper before Ant starts.

Source Code Editor or IDE

You need a text editor to edit Java source files, configuration files, HTML or XML pages, and Jakarta Server Pages (JSP). An editor that gracefully handles Windows and UNIX line-ending differences is preferred, but there are no other special requirements for your editor. You can edit HTML or XML pages and Jakarta Server Pages (JSP) with a plain text editor, or use a Web page editor such as Dreamweaver. For XML pages, you can also use an enterprise-level IDE with DTD validation or another development tool that supports editing of XML files.

Database System and JDBC Driver

Applications that access a database require an appropriate JDBC driver. If you configure a JDBC persistent store for JMS, use a database and driver supported for that store in this WebLogic Server release. JMS can also use a file persistent store. Consult Oracle Fusion Middleware Supported System Configurations for the release-specific database and driver matrix.

Web Browser

For browser-based applications, specify the browser versions and configurations that your application supports and test each of them. Include HTTPS and the TLS settings used by your

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

deployment in the test plan. Browser clients do not require the Java plug-in. Consult the release-specific certification information for any Oracle-provided browser application that you use.

Third-Party Software

You can use third-party software products to enhance your WebLogic Server development environment. WebLogic Developer Tools Resources provides developer tools information for products that support the application servers.

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

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

2

Using Ant Tasks to Configure and Use a WebLogic Server Domain

Learn about how to start and stop WebLogic Server instances and configure WebLogic Server domains using WebLogic Ant tasks in your development build scripts.

Overview of Configuring and Starting Domains Using Ant Tasks

WebLogic Server provides a pair of Ant tasks to help you perform common configuration tasks in a development environment. The configuration tasks enable you to start and stop WebLogic Server instances as well as create and configure WebLogic Server domains.

When combined with other WebLogic Ant tasks, you can create powerful build scripts for demonstrating or testing your application with custom domains. For example, a single Ant build script can:

- Compile your application using the `wlcompile`, `wlappc`, and Web services Ant tasks.
- Create a new single-server domain and start the Administration Server using the `wlserver` Ant task.
- Configure the new domain with required application resources using the `wlconfig` Ant task.
- Deploy the application using the `wldeploy` Ant task.
- Automatically start a compiled client application to demonstrate or test product features.

The sections that follow describe how to use the configuration Ant tasks, `wlserver` and `wlconfig`.

Starting Servers and Creating Domains Using the wlserver Ant Task

The `wlserver` Ant task enables you to start, reboot, shutdown, or connect to a WebLogic Server instance. The server instance may already exist in a configured WebLogic Server domain, or you can create a new single-server domain for development by using the `generateconfig=true` attribute.

For action="start" or action="reboot", wlserver waits for the server to reach RUNNING, subject to timeout or timeoutSeconds. action="connect" attempts a management connection to an already running server; it does not use the startup polling loop. By default, a server started by this task is shut down when the Ant build finishes. Set noExit="true" to leave it running. Connecting to a server started outside the build does not register it for automatic shutdown.

The wlserver task extends Ant's Java task and supports settings such as output and error for server output redirection. It also overrides some Java-task behavior: the server runs in a forked JVM; timeout and timeoutSeconds control the startup wait; and output redirection is ignored when noExit is true. See wlserver Ant Task Reference for these task-specific semantics and

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

the Apache Ant Java task reference for inherited settings. wlserver Ant Task Reference; Apache Ant Java task reference

> **Note**
>
> The default connection protocol is t3. To use a TLS-protected management endpoint, set protocol to t3s or https, use the port configured for that endpoint, and configure the Ant JVM to trust the server certificate. Use the endpoint required by the domain's administration-channel and secured-production-mode configuration.

Basic Steps for Using wlserver

To use the `wlserver` Ant task:

1. Set your environment.

   On Windows, execute the `setWLSEnv.cmd` command, located in the directory *`WL_HOME`*`\server\bin`, where *`WL_HOME`* is the top-level directory of your WebLogic Server installation.

   In a Bourne-compatible UNIX shell, source the environment script in the shell where you will run Ant:

   ```
   . "$WL_HOME/server/bin/setWLSEnv.sh"
   ```

   > **Note**
   >
   > The `wlserver` task is predefined in the version of Ant shipped with WebLogic Server. If you want to use the task with your own Ant installation, add the following task definition in your build file:
   >
   > ```
   > <taskdef name="wlserver" classname="weblogic.ant.taskdefs.management.WLServer"/>
   > ```

   > **Note**
   >
   > Use a supported Bourne-compatible shell, such as bash or Korn shell, and source the script using the dot command.

2. Add a call to the `wlserver` task in the build script to start, shutdown, restart, or connect to a server. See wlserver Ant Task Reference for information about `wlserver` attributes and default behavior.
3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   ```
   prompt> ant
   ```

   Use `ant -verbose` to obtain more detailed messages from the `wlserver` task.

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

Sample build.xml Files for wlserver

These are development examples. Supply WLS_USERNAME and WLS_PASSWORD through the environment used to launch Ant; for domain creation, the password must satisfy the domain's password policy. Add the following once under the build file's project element before these targets:

```
<property environment="env"/>
<fail unless="env.WLS_USERNAME" message="Set WLS_USERNAME before running
Ant."/>
<fail unless="env.WLS_PASSWORD" message="Set WLS_PASSWORD before running
Ant."/>
```

The following shows a minimal `wlserver` target that starts a server in the current directory using all default values:

```
<target name="wlserver-default">
  <wlserver username="${env.WLS_USERNAME}" password="${env.WLS_PASSWORD}"/>
</target>
```

This target connects to an existing, running server using the indicated connection parameters and user name/password combination:

```
<target name="connect-server">
  <wlserver username="${env.WLS_USERNAME}" password="${env.WLS_PASSWORD}"
host="127.0.0.1" port="7001" action="connect"/>
</target>
```

This target starts a WebLogic Server instance configured in the `config` subdirectory:

```
<target name="start-server">
  <wlserver username="${env.WLS_USERNAME}" password="${env.WLS_PASSWORD}" dir="./config"
host="127.0.0.1" port="7001" action="start"/>
</target>
```

This target creates a new single-server domain in an empty directory, and starts the domain's server instance:

```
<target name="new-server">
  <delete dir="./tmp"/>
  <mkdir dir="./tmp"/>
  <wlserver username="${env.WLS_USERNAME}" password="${env.WLS_PASSWORD}" dir="./tmp"
host="127.0.0.1" port="7001"
  generateConfig="true" action="start"/>
</target>
```

For existing domains, supply credentials for that domain and set servername to the configured server name when it differs from myserver.

These substitutions remove fixed secrets from the sample file; they do not conceal credentials from the server JVM command line, because explicit startup credentials are passed as JVM properties. These are not production credential-storage examples.

wlserver Ant Task Reference

The following table describes the attributes of the `wlserver` Ant task.

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

Table 2-1 Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
`policy`	The path to the security policy file for the WebLogic Server domain. This attribute is used only for starting server instances.	File	No
`dir`	The path that holds the domain configuration (for example, `c:\Oracle\Middleware\user_projects\domains\mydomain`). By default, `wlserver` uses the current directory.	File	No
`beahome`	The path to the Middleware Home directory (for example, `c:\Oracle\Middleware`).	File	No
`weblogichome`	The path to the WebLogic Server installation directory (for example, `c:\Oracle\Middleware\wlserver_12.1`).	File	No
`servername`	Name passed to the server process as weblogic.Name for start and reboot. The default is myserver; specify the configured name when starting a server with a different name. For connect and shutdown, the management endpoint identifies the target server. For server-naming restrictions, see Domain and Server Name Restrictions	String	No; set it for start/reboot when the configured name differs from myserver.
`domainname`	The name of the WebLogic Server domain in which the server is configured.	String	No
`adminserverurl`	The URL to access the Administration Server in the domain. This attribute is required if you are starting up a Managed Server in the domain.	String	Required for starting Managed Servers.
`username`	The user name of an administrator account. Supply this attribute for the Ant task's management connection. A server boot identity file does not supply credentials to the Ant task's JMX client. See Boot Identity Files in the Administering Server Startup and Shutdown for Oracle WebLogic Server for more information on boot.properties.	String	No
`password`	The password of an administrator account. Supply this attribute for the Ant task's management connection. A server boot identity file does not supply credentials to the Ant task's JMX client. See Boot Identity Files in the Administering Server Startup and Shutdown for Oracle WebLogic Server for more information on boot.properties.	String	No
`pkpassword`	The private key password for decrypting the SSL private key file.	String	No
`timeout`	Maximum startup polling wait for action="start" or action="reboot", in milliseconds. The default value for this attribute is 0, which means that the Ant task will wait indefinitely until the server transitions to the RUNNING state. This does not configure a retry wait for action="connect".	long	No
`timeoutSeconds`	Maximum startup polling wait for action="start" or action="reboot", in seconds. The default value for this attribute is 0, which means that the Ant task will wait indefinitely until the server transitions to the RUNNING state. This does not configure a retry wait for action="connect".	long	No

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

Table 2-1 (Cont.) Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
`productionmodeenabled`	Specifies whether a server instance boots in development mode or in production mode. Development mode enables a WebLogic Server instance to automatically deploy and update applications that are in the *`domain_name`*`/autodeploy` directory (where *`domain_name`* is the name of a WebLogic Server domain). In other words, development mode lets you use auto-deploy. Production mode disables the auto-deployment feature. See Deploying Applications and Modules for more information. Valid values for this attribute are True and False. The task attribute defaults to false. When true, wlserver passes -Dweblogic.ProductionModeEnabled=true to the server JVM. When false, it does not set that JVM property; the server's effective mode depends on the domain configuration and other startup settings. **Note:** If you boot the server in production mode by setting this attribute to `True`, you must reboot the server to set the mode back to development mode. Or in other words, you cannot reset the mode on a running server using other administrative tools, such as the WebLogic Server Scripting Tool (WLST).	Boolean	No
`host`	The DNS name or IP address on which the server instance is listening. The default value for this attribute is `localhost`.	String	No
`port`	The TCP port number on which the server instance is listening. The default value for this attribute is `7001`.	int	No
`generateconfig`	Specifies whether or not `wlserver` creates a new domain for the specified server. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No
`action`	Specifies the action `wlserver` performs: `start`, `shutdown`, `reboot`, or `connect`. The `shutdown` action can be used with the optional `forceshutdown` attribute to perform a forced shutdown. The default value for this attribute is `start`.	String	No
`failonerror`	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No

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

Table 2-1 (Cont.) Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?

`forceshutdown`	This optional attribute is used in conjunction with the `action="shutdown"` attribute to perform a forced shutdown. For example:
	
	`<wlserver`
	`host="${wls.host}"`
	`port="${port}"`
	`username="${wls.username}"`
	`password="${wls.password}"`
	`action="shutdown"`
	`forceshutdown="true"/>`
	
	`Valid values for this attribute are true and false. The default value is false.`
		Boolean	No

`noExit`	(Optional) Leave the server process running after the Ant build finishes. Valid values are `true` or `false`. The default value is `false`, which means the server process will shut down when the Ant build finishes.	Boolean	No
`protocol`	Specifies the protocol that the `wlserver` Ant task uses to communicate with the WebLogic Server instance. Valid values are `t3`, `t3s`, `http`, `https`, and `iiop`. The default value is `t3`.	String	No
`forceImplicitUpgrade`	Specifies whether the `wlserver` Ant task, if run against an 8.1 (or previous) domain, should implicitly upgrade it. Valid values are `true` or `false`. The default value is `false`, which means that the Ant task does *not* implicitly upgrade the domain, but rather, will fail with an error indicating that the domain needs to be upgraded. For more information about upgrading domains, see *Upgrading Oracle WebLogic Server*.	Boolean	No.
`configFile`	Specifies the configuration file for your domain. The value of this attribute must be a valid XML file that conforms to the XML schema as defined in the WebLogic Server Domain Configuration Schema at `http://xmlns.oracle.com/weblogic/domain/1.0/domain.xsd`. For an existing Administration Server domain, wlserver checks for the named configuration file under dir and then under dir/config. The default filename is config.xml. If dir is omitted, the base directory is the current directory. For a normal domain layout, retain the configuration at DOMAIN_HOME/config/config.xml and use DOMAIN_HOME as dir.	String	No.

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

Table 2-1 (Cont.) Attributes of the wlserver Ant Task

Attribute	Description	Data Type	Required?
`useBootProperties`	Specifies whether to use the `boot.properties` file when starting a WebLogic Server instance. If this attribute is set to `true`, WebLogic Server uses the user name and encrypted password stored in the `boot.properties` file to start rather than any values set with the `username` and `password` attributes. This setting applies to credentials passed to the server process during startup. It does not supply the credentials used by the Ant task to monitor startup, connect, shut down, or reboot a server; provide username and password for those management connections. Valid values are true and false. Default: false.	Boolean	No
`verbose`	Specifies that the Ant task output additional information as it is performing its action. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean	No

Configuring a WebLogic Server Domain Using the wlconfig Ant Task

You can use the `wlconfig` Ant task or the WebLogic Scripting Tool (WLST) to configure a WebLogic Server domain.

The following sections describe how to use the `wlconfig` Ant task to configure a WebLogic Server domain.

> **Note**
>
> For equivalent functionality, you should use the WebLogic Scripting Tool (WLST). See *Understanding the WebLogic Scripting Tool*.

What the wlconfig Ant Task Does

The `wlconfig` Ant task enables you to configure a WebLogic Server domain by creating, querying, or modifying configuration MBeans on a running Administration Server instance. Specifically, `wlconfig` enables you to:

- Create new MBeans, optionally storing the new MBean Object Names in Ant properties.
- Set attribute values on a named MBean available on the Administration Server.
- Create MBeans and set their attributes in one step by nesting set attribute commands within create MBean commands.
- Query MBeans, optionally storing the query results in an Ant property reference.
- Query MBeans and set attribute values on all matching results.

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

- Establish a parent/child relationship among MBeans by nesting create commands within other create commands.

Basic Steps for Using wlconfig

1. Set your environment in a command shell. See Basic Steps for Using wlserver for details.

> **Note**
>
> The `wlconfig` task is predefined in the version of Ant shipped with WebLogic Server. If you want to use the task with your own Ant installation, add the following task definition in your build file:

```
<taskdef name="wlconfig" classname="weblogic.ant.taskdefs.management.WLConfig"/>
```

2. `wlconfig` is commonly used in combination with `wlserver` to configure a new WebLogic Server domain created in the context of an Ant task. If you will be using `wlconfig` to configure such a domain, first use `wlserver` attributes to create a new domain and start the WebLogic Server instance.
3. Add an initial call to the `wlconfig` task to connect to the Administration Server for a domain. For example:

   After setting the WebLogic environment, supply admin.url, admin.userconfig and admin.userkey as Ant properties. The configuration/key files must already have been created with WLST storeUserConfig for the intended administrator account. This example queries the running Administration Server's edit MBean server and makes no configuration changes.

   ```
   <project name="wlconfig-example" default="doconfig">
     <taskdef name="wlconfig"
              classname="weblogic.ant.taskdefs.management.WLConfig"/>
     <target name="doconfig">
       <wlconfig url="${admin.url}"
                 userconfigfile="${admin.userconfig}"
                 userkeyfile="${admin.userkey}">
         <query type="Domain" property="domain.objectnames"/>
       </wlconfig>
       <echo message="Domain ObjectNames: ${domain.objectnames}"/>
     </target>
   </project>
   ```

   For a local development domain, admin.url can be t3://localhost:7001. Use the appropriate protected management URL for your environment. Extend this target with the nested operations described below.
4. Add nested `create`, `delete`, `get`, `set`, and `query` elements to configure the domain.
5. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   ```
   prompt> ant doconfig
   ```

   Use `ant -verbose` to obtain more detailed messages from the `wlconfig` task.

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

> **Note**
>
> Since WLST is the recommended tool for domain creation scripts, you should refer to the WLST offline sample scripts that are installed with the software. The offline scripts demonstrate how to create domains using the domain templates and are located in the following directory: *WL_HOME*\common\templates\scripts\wlst, where *WL_HOME* refers to the top-level installation directory for WebLogic Server. For example, the basicWLSDomain.py script creates a simple WebLogic domain, while sampleMedRecDomain.py creates a domain that defines resources similar to those used in the Avitek MedRec sample. See *Understanding the WebLogic Scripting Tool*.

wlconfig Ant Task Reference

The following sections describe the attributes and elements that can be used with wlconfig.

Main Attributes

The following table describes the main attributes of the wlconfig Ant task.

Table 2-2 Main Attributes of the wlconfig Ant Task

Attribute	Description	Data Type	Required ?
url	The URL of the domain's Administration Server.	String	Yes
username	The user name of an administrator account.	String	No
password	The password of an administrator account. To avoid having the plain text password appear in the build file or in process utilities such as ps, first store a valid user name and encrypted password in a configuration file using the WebLogic Scripting Tool (WLST) storeUserConfig command. Then omit both the username and password attributes in your Ant build file. When the attributes are omitted, wlconfig attempts to login using values obtained from the default configuration file. If you want to obtain a user name and password from a non-default configuration file and key file, use the userconfigfile and userkeyfile attributes with wlconfig. See the command reference for storeUserConfig in the *Understanding the WebLogic Scripting Tool* for more information on storing and encrypting passwords.	String	No
failonerror	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. This attribute is set to true by default.	Boolean	No

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

Table 2-2 (Cont.) Main Attributes of the wlconfig Ant Task

Attribute	Description	Data Type	Required ?
`userconfigfile`	Specifies the location of a user configuration file to use for obtaining the administrative user name and password. Use this option, instead of the `username` and `password` attributes, in your build file when you do not want to have the plain text password shown in-line or in process-level utilities such as `ps`. Before specifying the `userconfigfile` attribute, you must first generate the file using the WebLogic Scripting Tool (WLST) `storeUserConfig` command as described in the *Understanding the WebLogic Scripting Tool*.	File	No
`userkeyfile`	Specifies the location of a user key file to use for encrypting and decrypting the user name and password information stored in a user configuration file (the `userconfigfile` attribute). Before specifying the `userkeyfile` attribute, you must first generate the key file using the WebLogic Scripting Tool (WLST) `storeUserConfig` command as described in the *Understanding the WebLogic Scripting Tool*.	File	No

Nested Elements

`wlconfig` also has several elements that can be nested to specify configuration options:

- create
- delete
- set
- get
- query
- invoke

create

The `create` element creates a new MBean in the WebLogic Server domain. The `wlconfig` task can have any number of `create` elements.

A `create` element can have any number of nested `set` elements, which set attributes on the newly-created MBean. A `create` element may also have additional, nested `create` elements that create child MBeans.

The `create` element has the following attributes.

Table 2-3 Attributes of the create Element

Attribute	Description	Data Type	Required?
`name`	The name of the new MBean object to create.	String	No (`wlconfig` supplies a default name if none is specified.)

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

Table 2-3 (Cont.) Attributes of the create Element

Attribute	Description	Data Type	Required?
`type`	The MBean type.	String	Yes
`property`	Name of an optional Ant property that receives the ObjectName of the newly created MBean. It can be used on a top-level or nested create element.	String	No
realm	For security-provider creation, the name of the destination security realm. If omitted, the task uses the domain's default realm; nesting a fully qualified security-provider create under a realm create does not replace this selector.	String	No

delete

The `delete` element removes an existing MBean from the WebLogic Server domain. `delete` takes a single attribute:

Table 2-4 Attribute of the delete Element

Attribute	Description	Data Type	Required?
`mbean`	The object name of the MBean to delete.	String	Required when the `delete` element is a direct child of the `wlconfig` task. Not required when nested within a `query` element.

set

The `set` element sets MBean attributes on a named MBean, a newly-created MBean, or on MBeans retrieved as part of a query. You can include the `set` element as a direct child of the `wlconfig` task, or nested within a `create` or `query` element.

The `set` element has the following attributes:

Table 2-5 Attributes of the set Element

Attribute	Description	Data Type	Required?
`attribute`	The name of the MBean attribute to set.	String	Yes
`value`	The value to set for the specified MBean attribute. You can specify multiple object names (stored in Ant properties) as a value by delimiting the entire value list with quotes and separating the object names with a semicolon.	String	Yes
`mbean`	The object name of the MBean whose values are being set. This attribute is required only when the `set` element is included as a direct child of the main `wlconfig` task; it is not required when the set element is nested within the context of a `create` or `query` element.	String	Required only when the `set` element is a direct child of the `wlconfig` task.

Specify the complete target ObjectName, including its JMX domain, in mbean when set is a direct child of wlconfig. When nested in create or query, set receives its target from that parent.

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

get

The `get` element retrieves attribute values from an MBean in the WebLogic Server domain. The `wlconfig` task can have any number of `get` elements.

The `get` element has the following attributes.

Table 2-6 Attributes of the get Element

Attribute	Description	Data Type	Required?
`attribute`	The name of the MBean attribute whose value you want to retrieve.	String	Yes
`property`	The name of an Ant property that will hold the retrieved MBean attribute value.	String	Yes
`mbean`	ObjectName of the MBean whose attribute is read. A nested get reads the MBean currently selected by its parent query.	String	Required for a direct child of wlconfig; supplied by the enclosing query for a nested get.
mbeanserver	MBean server to read: edit, domainruntime or runtime. When omitted, a direct get uses edit and a nested get uses its query's server.	String	No

query

The `query` elements finds MBean that match a search pattern.

The query element supports the following nested child elements:

- `set`—performs set operations on all MBeans in the result set.
- `get`—performs get operations on all MBeans in the result set.
- `create`—each MBean in the result set is used as a parent of a new MBean.
- `delete`—performs delete operations on all MBeans in the result set.
- `invoke`—invokes all matching MBeans in the result set.

`wlconfig` can have any number of nested `query` elements.

`query` has the following attributes:

Table 2-7 Attributes of the query Element

Attribute	Description	Data Type	Required?
`domain`	The name of the WebLogic Server domain in which to search for MBeans.	String	No
`type`	The type of MBean to query.	String	No
`name`	The name of the MBean to query.	String	No
`pattern`	A JMX query pattern.	String	No
`property`	The name of an optional Ant property that will store the query results.	String	No

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

Table 2-7 (Cont.) Attributes of the query Element

Attribute	Description	Data Type	Required?
`domain`	This attribute specifies the JMX domain name for Security MBeans and third-party SPI MBeans. It is not required for administration MBeans, as the domain corresponds to the WebLogic Server domain.	String	No

invoke

The `invoke` element invokes a management operation for one or more MBeans. For WebLogic Server MBeans, you usually use this command to invoke operations other than the `get`*`Attribute`* and `set`*`Attribute`* that most WebLogic Server MBeans provide.

The `invoke` element has the following attributes.

Table 2-8 Attributes of the invoke Element

Attribute	Description	Data Type	Required?
`mbean`	The object name of the MBean you want to invoke.	String	You must specify either the `mbean` or `type` attribute of the invoke element.
`type`	The type of MBean to invoke.	String	You must specify either the `mbean` or `type` attribute of the invoke element.
`methodName`	The method of the MBean to invoke.	String	Yes
`arguments`	The list of arguments (separated by spaces) to pass to the method specified by the `methodName` attribute.	String	No

Example of Creating a Security Realm with the wlconfig Ant Task

You can use this example to create a security realm with the wlconfig Ant task:

Example 2-1 Creating a Security Realm with wlconfig

This is an online configuration edit through the Administration Server. Run it within an Ant target after defining wlconfig. Supply admin.url, admin.userconfig and admin.userkey as in the basic example. MyRealm must not already exist. The task activates the changes when it completes; the example creates an additional realm and leaves the current default realm selected. It does not configure users or demonstrate switching the domain's active realm.

```
<wlconfig url="${admin.url}"
          userconfigfile="${admin.userconfig}"
          userkeyfile="${admin.userkey}">
  <create type="weblogic.management.security.Realm"
          name="MyRealm" property="new.realm">
    <create name="MyAuthenticator" realm="MyRealm"
            type="weblogic.security.providers.authentication.DefaultAuthenticator"/>
    <create name="MyAuthorizer" realm="MyRealm"
            type="weblogic.security.providers.authorization.DefaultAuthorizer"/>
    <create name="MyRoleMapper" realm="MyRealm"
            type="weblogic.security.providers.authorization.DefaultRoleMapper"/>
    <create name="MyCredentialMapper" realm="MyRealm"
```


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

```
            type="weblogic.security.providers.credentials.DefaultCredentialMapper"/>
    <create name="MyCertPathProvider" realm="MyRealm"
            type="weblogic.security.providers.pk.WebLogicCertPathProvider"
            property="new.certpath"/>
  </create>
  <set mbean="${new.realm}" attribute="CertPathBuilder"
       value="${new.certpath}"/>
</wlconfig>
```

Using the libclasspath Ant Task

Use the `libclasspath` Ant task to build applications that use libraries, such as application libraries and Web libraries.

libclasspath Task Definition

To use the task with your own Ant installation, add the following task definition in your build file:

```
<taskdef name="libclasspath" classname="weblogic.ant.taskdefs.build.LibClasspathTask"/>
```

libclasspath Ant Task Reference

The following sections describe the attributes and elements that can be used with the `libclasspath` Ant task.

- Main libclasspath Attributes
- Nested libclasspath Elements

Main libclasspath Attributes

The following table describes the main attributes of the `libclasspath` Ant task.

Table 2-9 Attributes of the libclasspath Ant Task

Attribute	Description	Required
`basedir`	The root of `.ear` or `.war` file to extract from.	Either `basedir` or `basewar` is required.
`basewar`	The name of the `.war` file to extract from.	If `basewar` is specified, `basedir` is ignored and the library referenced in `basewar` is used as the `.war` file to extract classpath or resourcepath information from.
`tmpdir`	The fully qualified name of the directory to be used for extracting libraries.	Yes.
`classpathproperty`	Contains the classpath for the referenced libraries. For example, if `basedir` points to a `.war` file that references Web application libraries in the `weblogic.xml` file, the `classpathproperty` contains the `WEB-INF/classes` and `WEB-INF/lib` directories of the Web application libraries. Additionally, if `basedir` points to a `.war` file that has `.war` files under `WEB-INF/bea-ext`, the `classpathproperty` contains the `WEB-INF/classes` and `WEB-INF/lib` directories for the Oracle extensions.	At least one of the two attributes is required.

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

Table 2-9 (Cont.) Attributes of the libclasspath Ant Task

Attribute	Description	Required
`resourcepathproperty`	Contains library resources that are not classes. For example, if `basedir` points to a `.war` file that has `.war` files under `WEB-INF/bea-ext`, `resourcepathproperty` contains the roots of the exploded extensions.	

Nested libclasspath Elements

`libclasspath` also has two elements that can be nested to specify configuration options. At least one of the elements is required when using the `libclasspath` Ant task:

librarydir

The following attribute is required when using this element:

`dir`—Specifies that all files in this directory are registered as available libraries.

library

The following attribute is required when using this element:

`file`—Register this file as an available library.

Example libclasspath Ant Task

This section provides example code of a `libclasspath` Ant task:

Example 2-2 Example libclasspath Ant Task Code

Insert this taskdef and target fragment under the Ant project element.

```
<taskdef name="libclasspath" classname="weblogic.ant.taskdefs.build.LibClasspathTask"/>

   <!-- Builds classpath based on libraries defined in weblogic-application.xml. -->
   <target name="init.app.libs">
      <libclasspath basedir="${src.dir}" tmpdir="${tmp.dir}"
classpathproperty="app.lib.classpath">
         <librarydir dir="${weblogic.home}/common/deployable-libraries/"/>
      </libclasspath>
   <echo message="app.lib.classpath is ${app.lib.classpath}" level="info"/>
   </target>
```


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

3

Using the WebLogic Maven Plug-In

Apache Maven is a software tool for building and managing Java-based projects. WebLogic Server provides support for Maven through the provisioning of plug-ins that enable you to perform various operations on WebLogic Server from within a Maven environment.
The `weblogic-maven-plugin` provides enhanced functionality to install, start and stop servers, create domains, execute WLST scripts, and compile and deploy applications. With the `weblogic-maven-plugin`, you can install WebLogic Server from within your Maven environment to fulfill the local WebLogic Server requirement when needed.

See Building Java EE Projects for WebLogic Server with Maven in *Developing Applications Using Continuous Integration* for additional Maven documentation.

Installing Maven

To use the `weblogic-maven-plugin` plug-in, you must first have a functional Maven installation and a Maven repository.

WebLogic Server and Fusion Middleware 26.1.0 support Maven 3.9.4 and later versions. A distribution of Maven is included with Fusion Middleware 26.1.0.0.0 in the following location: `ORACLE_HOME\oracle_common\modules\thirdparty\apache-maven_bundle\3.9.4.0.0`. This is a copy of the standard Maven release, without any modifications. For information about the specific version of Maven that is included in WebLogic Server, see *Third-Party Products* in *Oracle Fusion Middleware in Oracle® Fusion Middleware Licensing Information User Manual*. You can download and install your own copy of Maven from the Maven Web site: `http://maven.apache.org.` Make sure you set any required variables as detailed in that documentation, such as `M2_HOME` and `JAVA_HOME`.

> **Note**
>
> Maven is not included in the 26.1.0.0.0 in the WebLogic Server distribution.

Run the *`ORACLE_HOME`*`\wlserver\server\bin\setWLSEnv` script to configure Maven.

> **Note**
>
> The `weblogic-maven-plugin` sets the Java protocol handler to `weblogic.net`. To use the default JDK protocol handlers, specify the system property `-DUseSunHttpHandler=true` in the JVM that executes Maven. To do this, override the environment variable `MAVEN_OPTS` inside the `mvn.bat` or `mvn.sh` files to set the appropriate value. For example: `set MAVEN_OPTS="-DUseSunHttpHandler=true"`.

For detailed information on installing and using Maven to build applications and projects, see the Maven Users Centre at `http://maven.apache.org/users/index.html`.

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

Configuring the WebLogic Maven Plug-In

Use the pre-built JAR file and accompanying POM file to install and configure `weblogic-maven-plugin`.

Complete the following steps to install and configure `weblogic-maven-plugin`:

1. Install the Oracle Maven sync plug-in and run the push goal:
   a. Change the directory to: *ORACLE_HOME*`\oracle_common\plugins\maven\com\oracle\maven\oracle-maven-sync\26.1.0`
   b. `mvn install:install-file -DpomFile=oracle-maven-sync-26.1.0.pom -Dfile=oracle-maven-sync-26.1.0.jar`
   c. `mvn com.oracle.maven:oracle-maven-sync:push -DoracleHome=c:\oracle\middleware\oracle_home\`
2. To validate successful installation of the plug-in, use the Maven `help:describe` goal. For more information, see the Apache help plug-in describe goal documentation.

```
mvn help:describe -DgroupId=com.oracle.weblogic
-DartifactId=weblogic-maven-plugin -Dversion=26.1.0-0-0
```

How to use the WebLogic Maven Plug-in

There are two ways to invoke the goals in the WebLogic Maven plug-in:

- From a Maven project POM.
- From the command line.

The `appc`, `wsgen`, `wsimport`, `ws-jwsc`, `ws-wsdlc`, and `ws-clientgen` goals require a POM.

Other goals will work either way. For example, `install`, `wlst`, `wlst-client`, `start-server`, or `stop-server` work either from a POM or the command line.

The preferred and recommended way is to use a Maven POM file.

To invoke a WebLogic Maven plug-in goal from a POM file, do the following:

1. Add a build section to your POM if you do not already have one.
2. Add a plug-in section to the build section for the WebLogic Maven plug-in.
3. Add an execution section to the WebLogic Maven plug-in's `plugin` section for each goal that you want to execute. This section must provide the necessary parameters for the goal, and map the goal to a phase in the Maven Lifecycle.

The following shows an example of the necessary additions, including a few goals. The detailed descriptions of each goal later in this section present the details for parameters and examples for each goal.

If you map multiple goals to the same lifecycle phase, they are typically executed in the order you list them in the POM.

Example 3-1 Modifying the POM File

```
<build>
    <plugins>
      <plugin>
```


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

```
        <!-- This is the configuration for the
             weblogic-maven-plugin
        -->
        <groupId>com.oracle.weblogic</groupId>
<artifactId>weblogic-maven-plugin</artifactId>
        <version>26.1.0-0-0</version>
        <configuration>
<middlewareHome>/fmwhome/wls26100</middlewareHome>
        </configuration>
        <executions>
          <!-- Execute the appc goal during the package phase -->
          <execution>
            <id>wls-appc</id>
            <phase>package</phase>
            <goals>
              <goal>appc</goal>
            </goals>
            <configuration>
<source>${project.build.directory}/${project.name}.${project.packaging}</source>
            </configuration>
          </execution>
          <!-- Deploy the application to the WebLogic Server in the
               pre-integration-test phase
          -->
          <execution>
            <id>wls-deploy</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>deploy</goal>
            </goals>
            <configuration>
              <!--The admin URL where the app is deployed.
Here use the plugin's default value t3://localhost:7001-->
<adminurl>t3://127.0.0.1:7001</adminurl>
              <user>weblogic</user>
              <password>password</password>
              <!--The location of the file or directory to be deployed-->
<source>${project.build.directory}/${project.build.finalName}.${project.packaging}</
source>
              <!--The target servers where the application is deployed.
Here use the plugin's default value AdminServer-->
              <targets>AdminServer</targets>
              <verbose>true</verbose>
<name>${project.build.finalName}</name>
            </configuration>
          </execution>
          <!-- Stop the application in the pre-integration-test phase -->
          <execution>
            <id>wls-stop-app</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>stop-app</goal>
            </goals>
            <configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
              <user>weblogic</user>
              <password>password</password>
<name>${project.build.finalName}</name>
            </configuration>
          </execution>
        </executions>
      </plugin>
```


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

```
    </plugins>
  </build>
```

Table 3-1 lists the phases in the default Maven life cycle.

Table 3-1 Maven Lifecycle Phases

Phase	Description
validate	Validates the project is correct and all necessary information is available.
compile	Compiles the source code of the project.
test	Tests the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
package	Takes the compiled code and package it in its distributable format, such as a JAR.
integration-test	Processes and deploys the package if necessary into an environment where integration tests can be run.
verify	Runs any checks to verify the package is valid and meets quality criteria.
install	Installs the package into the local repository, for use as a dependency in other projects locally.
deploy	In an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Table 3-2 shows the most common mappings of goals to phases

Table 3-2 Common Mapping of Goals to Phases

Phase	Goal
validate	`ws-clientgen, ws-wsdlc`
compile	`ws-jwsc`
test	NA
package	`appc`
pre-integration-test[1]	`install, create-domain, start-server, distribute-app, deploy, purge-tasks, redeploy, update-app, start-app, stop-app, wlst, wlst-client,` and `list-apps`
post-integration-test[2]	`remove-domain, undeploy, stop-server, uninstall`
verify	NA
install	NA
deploy	NA

1 The integration-test phase has pre sub-phases that are executed before the actual execution of any integration tests, respectively.
2 The integration-test phase has post sub-phases that are executed after the actual execution of any integration tests, respectively.

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

Basic Configuration POM File

Example 3-2 illustrates a basic Jakarta EE web application pom.xml file that demonstrates the use of the weblogic-maven-plugin appc goal.

Example 3-2 Basic Configuration pom.xml File

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.sab</groupId>
    <artifactId>maven-demo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>maven-demo</name>

    <properties>
        <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
      <dependency>
       <groupId>com.oracle.weblogic</groupId>
       <artifactId>weblogic-server-pom</artifactId>
       <version>26.1.0-0-0</version>
       <type>pom</type>
       <scope>provided</scope>
      </dependency>
    </dependencies>

    <build>
        <plugins>
            ...
            ...

            <!-- WebLogic Server 26 Maven Plugin -->
            <plugin>
                <groupId>com.oracle.weblogic</groupId>
                <artifactId>weblogic-maven-plugin</artifactId>
                <version>15.1.1-0-0</version>
            </plugin>
            <configuration>
             </configuration>
             <executions>
               <execution>
                <id>wls-appc</id>
                <phase>package</phase>
                <goals>
                 <goal>appc</goal>
                </goals>
               <configuration>
                 <source>${project.build.directory}/${project.name}.
                   ${project.packaging}</source>
               </configuration>
              </execution>
             </executions>
```


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

```
            </plugins>
        </build>

</project>
```

Maven Plug-In Goals

See an alphabetical listing of all the Maven plug-in goals.

Table 3-3 lists all the `weblogic-maven-plugin` goals. Each goal is described in detail in the sections that follow.

Table 3-3 Maven Plug-In Goals

Goal Name	Description
appc	Generates and compiles the classes needed to deploy EJBs and JSPs to WebLogic Server. Also validates the deployment descriptors for compliance with the current specifications at both the individual module level and the application level. Does not require a local server installation.
create-domain	Creates a domain for WebLogic Server using a domain template. This goal supports specifying the domain directory (the last directory determines the domain name) and the administrative username and password. For more complex domain creation, use the `wlst` goal.
deploy	Deploys WebLogic Server applications and modules to a running server. Supports all deployment formats; for example, WAR, JAR, RAR, and such.
distribute-app	Prepares deployment files for deployment by copying deployment files to target servers and validating them.
install	Installs WebLogic Server.
list-apps	Lists the deployment names for applications and standalone modules deployed, distributed, or installed in the domain.
purge-tasks	Flushes out retired deployment tasks.
redeploy	Redeploys a running application or part of a running application.
remove-domain	Removes a domain directory.
start-app	Starts an application deployed on WebLogic Server.
start-server	Starts WebLogic Server. This goal starts WLS by running a local start script. For starting remote servers using the node manager, use the wlst goal instead.
stop-app	Stops an application.
stop-server	Stops WebLogic Server. This goal stops WLS by running a local start script. For stopping remote servers using the node manager, use the wlst goal instead.
undeploy	Undeploys the application from WebLogic Server. Stops the deployment unit and removes staged files from target servers.
uninstall	Uninstalls WebLogic Server.
update-app	Updates an application's deployment plan by redistributing the plan files and reconfiguring the application based on the new plan contents.
wlst	WLST wrapper for Maven.
wlst-client	WLST wrapper that does not require a local server install for WLST online commands.
ws-clientgen	Generates client Web service artifacts from a WSDL.

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

Table 3-3 (Cont.) Maven Plug-In Goals

Goal Name	Description
wsgen	JAX-WS service endpoint implementation class and generates all of the portable artifacts for a JAX-WS Web service.
wsimport	Maven goal that parses a WSDL and binding files and generates the Java code needed to access it
ws-jwsc	Builds a JAX-WS Web service.
ws-wsdlc	Generates a set of artifacts and a partial Java implementation of the Web service from a WSDL.

appc

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:appc`

Description

Generates and compiles the classes needed to deploy EJBs and JSPs to WebLogic Server. Also validates the deployment descriptors for compliance with the current specifications at both the individual module level and the application level. Does not require a local server installation.

Parameters

Table 3-4 appc Parameters

Name	Type	Required	Description
`altappdd`	`java.lang.String`	false	Specifies an alternate descriptor. May be used to specify an alternate `application.xml` for an `.ear` deployment or an alternate `web.xml` or `ejb-jar.xml` for standalone module deployments.
`altwlsappdd`	`java.lang.String`	false	Specifies the path to an alternative WebLogic Server application deployment descriptor.
`basicClientJar`	`boolean`	false	When true, does not include deployment descriptors in client JARs generated for EJBs. Default value is: `false`
`classpath`	`java.lang.String`	false	This parameter is deprecated in this release and ignored. Use the standard Maven dependency model instead to manipulate the effective CLASSPATH during a build.
`clientJarOutputDir`	`java.lang.String`	false	Specifies a directory where generated client JARs will be written.
`commentary`	`boolean`	false	This parameter is deprecated in this release.
`compiler`	`java.lang.String`	false	Specifies the Java compiler for compiling class files from the generated Java source code. The Java compiler program should be in your PATH unless you specify the absolute path to the compiler explicitly. Default value is: `javac`
`compilerClass`	`java.lang.String`	false	The class that invokes the compiler. Default value is: `com.sun.tools.javac.Main`

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

Table 3-4 (Cont.) appc Parameters

Name	Type	Required	Description
continueCompilation	boolean	false	When true, continues compilation even when there are errors in the JSP files. Default value is: false
debug	boolean	false	When true, compiles debugging information into class files. Default value is: false
deprecation	boolean	false	When true, warns about the use of deprecated methods in the generated Java source file when compiling the source file into a class file. Default value is: false
destdir	java.lang.String	false	Specifies the directory where compiled class files are written. Use this parameter to place compiled classes in a directory that is already in your CLASSPATH.
enableHotCodeGen	boolean	false	This parameter is deprecated in this release.
forceGeneration	boolean	false	When true, forces the generation of EJB and JSP classes. Otherwise, the classes will not be regenerated if it is determined to be unnecessary. Default value is: false
idl	boolean	false	When true, generates IDL for EJB remote interfaces. Default value is: false
idlDirectory	java.lang.String	false	Specifies the directory where IDL files will be written. Default: the target directory or JAR
idlFactories	boolean	false	When true, generates factory methods for valuetypes. Default value is: false
idlMethodSignatures	java.lang.String	false	Specifies the method signatures used to trigger IDL code generation.
idlNoAbstractInterfaces	boolean	false	When true, does not generate abstract interfaces and methods or attributes that contain them. Default value is: false
idlNoValueTypes	boolean	false	Does not generate valuetypes or the methods and attributes that contain them. Default value is: false
idlOrbix	boolean	false	When true, generates IDL somewhat compatible with Orbix C++. Default value is: false
idlOverwrite	boolean	false	When true, overwrites existing IDL files. Default value is: false
idlVerbose	boolean	false	When true, displays additional status information for IDL generation. Default value is: false
idlVisibroker	boolean	false	When true, generates IDL somewhat compatible witih Visibroker C++. Default value is: false
ignorePlanValidation	boolean	false	When true, ignores the plan file if it does not exist.
iiop	boolean	false	When true, generates CORBA stubs for EJBs. Default value is: false
iiopDirectory	java.lang.String	false	Specifies the directory where IIOP stub files will be written. Default: the target directory or JAR
keepgenerated	boolean	false	When true, preserves the generated .java files. Default value is: false
libraries	java.lang.String	false	A comma-separated list of libraries.

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

Table 3-4 (Cont.) appc Parameters

Name	Type	Required	Description
`librarydir`	java.lang.String	false	Registers all the files in the specified directory as libraries.
`lineNumbers`	`boolean`	false	When true, adds JSP line numbers to generated class files to aid in debugging. Default value is: `false`
`manifest`	`java.io.File`	false	This parameter is deprecated in this release. Use the standard Maven mechanism to specify the Manifest during packaging.
`maxfiles`	`java.lang.Integer`	false	Specifies the maximum number of generated Java files to be compiled at one time.
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`noexit`	`boolean`	false	When true, appc throws an exception instead of terminating the JVM with System.exit when it encounters an error. Keep this enabled when running appc inside Maven. Default: true. To continue compiling JSP files after individual compilation errors, see continueCompilation.
`normi`	`boolean`	false	This parameter is deprecated in this release.
`nowarn`	`boolean`	false	When true, suppresses compiler warnings. Default value is: `false`
`nowrite`	`boolean`	false	This parameter is deprecated in this release.
`optimize`	`boolean`	false	When true, compiles with optimization on. Default value is: `false`
`output`	java.lang.String	false	Specifies an alternate output archive or directory. When not set, the output is placed in the source archive or directory.
`plan`	java.lang.String	false	Specifies the path to an optional deployment plan.
`quiet`	`boolean`	false	When true, turns off output except for errors.
`runtimeFlags`	`java.lang.String`	false	Passes a list of options to the compiler.
`serverClasspath`	`java.lang.String`	false	This parameter is deprecated in this release and ignored. Use the standard Maven dependency model instead to manipulate the effective CLASSPATH.
`source`	java.lang.String	false	Specifies the path to the source files. Default value is: ${project.build.directory}/${project.artifactId}.${project.packaging} For an archive produced by Maven packaging, explicitly configure source as ${project.build.directory}/${project.build.finalName}.${project.packaging}; the plugin default does not include Maven's usual version suffix.
`sourceVersion`	`java.lang.String`	false	Specifies the Java source-language version for generated Java compilation, for example, 21. The default value is the JDK version of the Java compiler used. Use levels supported by the selected compiler and the deployment runtime; sourceVersion does not by itself restrict references to newer Java APIs.
`supressCompiler`	`boolean`	false	This parameter is deprecated in this release and ignored. Use the standard Maven dependency model instead to add the target classes to the effective CLASSPATH during a build.

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

Table 3-4 (Cont.) appc Parameters

Name	Type	Required	Description
targetVersion	java.lang.String	false	Specifies the class-file target version for generated Java compilation, for example, 21. The default value is the JDK version of the Java compiler used. Use levels supported by the selected compiler and the deployment runtime; sourceVersion does not by itself restrict references to newer Java APIs.
verbose	boolean	false	When true, displays additional status information during the compilation process. Default value is: false
verboseJavac	boolean	false	When true, enables verbose output from the Java compiler. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.
writeInferredDescriptors	boolean	false	When true, writes out the descriptors with inferred information including annotations.

Usage Example

The appc goal executes the WebLogic Server application compiler utility to prepare an application for deployment.

```
<execution>
<id>wls-appc</id>
<phase>package</phase>
<goals>
<goal>appc</goal>
</goals>
<configuration>
<source>${project.build.directory}/${project.build.finalName}.${project.packaging}</
source>
</configuration>
</execution>
```

Example 3-3 shows typical appc goal output.

Example 3-3 appc

```
$ mvn com.oracle.weblogic:weblogic-maven-plugin:appc
 -Dsource=target/basicWebapp.war -DforceGeneration=true
```

Illustrative output from an earlier documentation example; release banners, timestamps, paths and generated files can differ. This output is not a verification run for WebLogic Server 26.1.0.0.0.

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building basicWebapp 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:15.1.1-0-0:appc (default-cli) @ main-test ---
[INFO] Running weblogic.appc on
/home/oracle/src/tests/main-test/target/basicWebapp.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```


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

```
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.901s
[INFO] Finished at: Wed Aug 19 10:52:46 EST 2015
[INFO] Final Memory: 26M/692M
[INFO]
```

create-domain

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:create-domain`

Description

Creates a domain for WebLogic Server using a domain template. This goal supports specifying the domain directory (the last directory determines the domain name) and the administrative username and password. For more complex domain creation, use the `wlst` goal.

> **Note**
>
> Beginning in version 12.2.1, there is a single unified version of WLST that automatically includes the WLST environment from all products in the `ORACLE_HOME`.

Parameters

Table 3-5 create-domain Parameters

Name	Type	Required	Description
`domainHome`	`java.lang.String`	true	Specifies the directory to use for creating the domain. This goal takes the name of the last subdirectory specified as the domain name and sets the new domain's name to that value. For example, domainHome=/weblogic/domains/MyNewDomain causes the domain name to be set to 'MyNewDomain'.
`domainTemplate`	`java.lang.String`	false	Specifies the domain template file to use to create the domain. The default domain template included with WebLogic Server is used when this parameter is not specified.
`failOnDomainExists`	`boolean`	false	When `true` and the domain to be created already exists, the build fails and an exception is thrown. When `false` and the domain to be created already exists, the build is successful and the existing domain is not overwritten. If the domain does not exist, this parameter has no effect. Default value is: `false`
`middlewareHome`	`java.lang.String`	true	The path to the Oracle Middleware install directory.
`password`	`java.lang.String`	**true**	Specifies the administrative password.
`serverClasspath`	`java.lang.String`	false	This parameter is deprecated and ignored in this release.
`user`	`java.lang.String`	**true**	Specifies the administrative user name.
`weblogicHome`	`java.lang.String`	false	This parameter is deprecated and ignored in this release.

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

Table 3-5 (Cont.) create-domain Parameters

Name	Type	Required	Description
wlstVersion	java.lang.String	false	Deprecated. As of version 12.2.1, there is a single, unified version of WLST. This parameter is deprecated and ignored.
workingDir	java.lang.String	false	The current working directory where the create-domain goal executes. The default value is: ${project.build.directory}/weblogic-maven-plugin

Usage Example

Use the `create-domain` goal to create a WebLogic Server domain from a specified WebLogic Server installation. You specify the location of the domain using the `domainHome` configuration parameter.

When creating a domain, a user name and password are required. You can specify these using the `user` and `password` configuration parameters in your POM file or by specifying them on the command line.

The domain name is taken from the last subdirectory specified in `domainHome`.

```
<execution>
<id>wls-create-domain</id>
<phase>pre-integration-test</phase>
<goals>
<goal>create-domain</goal>
</goals>
<configuration>
<middlewareHome>c:/dev/wls261000</middlewareHome>
<domainHome>${project.build.directory}/base_domain</domainHome>
<user>weblogic</user>
<password>password</password>
</configuration>
</execution>
```

Example 3-4 shows typical command output from the execution of the `create-domain` goal.

Example 3-4 create-domain

```
mvn com.oracle.weblogic:weblogic-maven-plugin:create-domain
-DdomainHome=c:\oracle\middleware\oracle_home\user_projects\domains\maven-domain
-DmiddlewareHome=c:\oracle\middleware\oracle_home -Duser=weblogic -Dpassword=password
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0-0-0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:15.1.1-0-0:create-domain (default-cli) @
 weblogic-maven-plugin ---
[INFO] [create-domain]Domain creation script:
readTemplate(r'C:/oracle/middleware/oracle_home/wlserver/common/templates/wls/wls.jar')
set('Name', 'maven-domain')
cd('/Security/maven-domain/User/weblogic')
set('Name', 'weblogoc')
set('Password', '***')
writeDomain(r'c:/oracle/middleware/oracle_home/user_
projects/domains/maven-domain')
[INFO] [wlst]script temp file = C:/Users/user/AppData/Local/Temp/
```


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

```
test6066166061714573929.py
[INFO] [wlst]Executing: [cmd:[C://windows\\system32\\cmd.exe, /c,
C:\oracle\middleware\oracle_home\wlserver\common\bin\wlst.cmd
C:\Users\user\AppData\Local\Temp\test6066166061714573929.py ]]
[INFO] Process being executed, waiting for completion.
[INFO] [exec]
[INFO] [exec] Initializing WebLogic Scripting Tool (WLST) ...
[INFO] [exec]
[INFO] [exec] Welcome to WebLogic Server Administration Scripting Shell
[INFO] [exec]
[INFO] [exec] Type help() for help on available commands
[INFO] [exec]
[INFO] [wlst][cmd:[C:\\windows\\system32\\cmd.exe, /c,
C:\oracle\middleware\oracle_home\wlserver\common\bin\wlst.cmd
C:\Users\user\AppData\Local\Temp\test6066166061714573929.py ]] exit code=0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 18.276s
[INFO] Finished at: Wed Aug 19 13:13:25 EDT 2015
[INFO] Final Memory: 9M/23M
[INFO] ------------------------------------------------------------------------
```

deploy

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:deploy`

Description

Deploys WebLogic Server applications and modules to a running server. Supports all deployment formats; for example, WAR, JAR, RAR, and such. Does not require a local server installation.

Parameters

Table 3-6 deploy Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: t3://localhost:7001
`advanced`	`boolean`	false	When true, prints advanced usage options.
`altappdd`	`java.lang.String`	false	Specifies an alternate descriptor. May be used to specify an alternate application.xml for an .ear deployment or an alternate web.xml or ejb.xml for standalone module deployments.
`appversion`	`java.lang.String`	false	Version of the application to start.
`debug`	`boolean`	false	When true, displays debug-level messages to the standard output. Default value is: false
`enableSecurityValidation`	`boolean`	false	When true, enables validation of security data. Default value is: false
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.

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

Table 3-6 (Cont.) deploy Parameters

Name	Type	Required	Description
external_stage	boolean	false	When true, indicates that the user wants to copy the application in the server staging area externally or using a third-party tool. When specified, WebLogic Server looks for the application under StagingDirectoryName(of target server)/applicationName. Default value is: false
failOnError	boolean	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: true
id	java.lang.String	false	Specifies an optional, user-supplied, unique deployment task identifier.
libimplver	java.lang.String	false	Implementation version of a Jakarta EE library or optional package. This option can be used only if the library or package does not include an implementation version in its manifest file.
library	boolean	false	Deploy as a shared Jakarta EE library or optional package.
libspecver	java.lang.String	false	Specification version of a Jakarta EE library or optional package. This option can be used only if the library or package does not include a specification version in its manifest file.
middlewareHome	java.lang.String	false	This parameter is deprecated in this release and ignored.
name	java.lang.String	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.
nostage	boolean	false	When true, does not copy the deployment files to target servers, but leaves them in a fixed location, specified by the source parameter. By default, nostage is true for the Administration Server and stage is true for the Managed Server targets.
noversion	boolean	false	When true, ignores all version related code paths on the Administration Server. Default value is: false
nowait	boolean	false	When true, initiates multiple tasks and then monitors them later with the -list action. Default value is: false
password	java.lang.String	false	Specifies the administrative password.
plan	java.lang.String	false	Specifies the path to the deployment plan.
remote	boolean	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the source parameter specifies a path on the server, unless the upload parameter is also used. Default value is: false
retiretimeout	java.lang.Integer	false	Specifies the number of seconds before WebLogic Server undeploys the currently-running version of this application or module so that clients can start using a new version. When not specified, a graceful retirement policy is assumed. Default value is: -1
securityModel	java.lang.String	false	Specifies the security model to be used for this deployment, overriding the default security model for the security realm. Possible values are: DDOnly, CustomRoles, CustomRolesAndPolicies, and Advanced.
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.

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

Table 3-6 (Cont.) deploy Parameters

Name	Type	Required	Description
source	java.lang.String	false	Specifies the address of the artifact to deploy. The address can be one of the following: • A colon (:) separated list of Maven coordinates of the form: groupId:artifactId:packaging:classifier:version. • An archive file or exploded archive directory on the local system. For example, /home/myhome/myapps/helloworld.war. • A remote HTTP URL (http://foo/a/b.ear).
stage	boolean	false	When true, indicates that the application needs to be copied into the target server staging area before deployment. By default, nostage is true for the Administration Server and stage is true for the Managed Server targets.
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	Specifies a comma-separated list of targets for the current operation. The default is AdminServer.
timeout	java.lang.Integer	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1
upload	boolean	false	When true, copies the source files to the Administration Server's upload directory prior to deployment. Use this setting when running the plug-in remotely (using the remote parameter) and when the user lacks normal access to the Administration Server's file system. Default value is: false.
usenonexclusivelock	boolean	false	When true, the deployment operation uses an existing lock, already acquired by the same user, on the domain. This parameter is helpful in environments where multiple deployment tools are used simultaneously and one of the tools has already acquired a lock on the domain configuration. Default value is: false.
user	java.lang.String	false	Specifies the administrative user name.
userConfigFile	java.lang.String	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
userKeyFile	java.lang.String	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
verbose	boolean	false	When true, displays additional status information. Default value is: false
version	boolean	false	When true, prints the version information. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

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

Usage Example

Use this goal to deploy an application.

```
<execution>
<id>wls-deploy</id>
<phase>pre-integration-test</phase>
<goals>
<goal>deploy</goal>
</goals>
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<source>${project.build.directory}/${project.build.finalName}
.${project.packaging}</source>
<targets>AdminServer</targets>
<verbose>true</verbose>
<name>${project.build.finalName}</name>
</configuration>
</execution>
```

Example 3-5 shows typical `deploy` goal output.

Example 3-5 deploy

```
mvn com.oracle.weblogic:weblogic-maven-plugin:deploy
-Dsource=C:\webservices\MySimpleEjb.jar
-Dpassword=password -Duser=weblogic
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0-0-0
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:15.1.1-0-0:deploy (default-cli) @ weblogic-
mave
n-plugin ---
weblogic.Deployer invoked with options:  -noexit -adminurl t3://
localhost:7001 -
deploy -user weblogic -source C:\webservices\MySimpleEjb.jar -targets
AdminServe
r
<Aug 19, 2015> <Info> <J2EE Deployment SPI> <BEA-260121> <Initiati
ng deploy operation for application, MySimpleEjb [archive:
C:\webservices\MySimp
leEjb.jar], to AdminServer .>
Task 0 initiated: [Deployer:149026]deploy application MySimpleEjb on
AdminServer
.
Task 0 completed: [Deployer:149026]deploy application MySimpleEjb on
AdminServer
.
```


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

```
Target state: deploy completed on Server AdminServer

[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 9.042s
[INFO] Finished at: Wed Aug 19 13:41:11 EDT 2015
[INFO] Final Memory: 10M/25M
```

distribute-app

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:distribute-app`

Description

Prepares deployment files for deployment by copying deployment files to target servers and validating them. Does not require a local server installation.

Parameters

Table 3-7 distribute-app Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: t3://localhost:7001
`advanced`	`boolean`	false	When true, prints advanced usage options.
`debug`	`boolean`	false	When true, displays debug-level messages to the standard output. Default value is: false
`enableSecurityValidation`	`boolean`	false	When true, enables validation of security data. Default value is: false
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.
`external_stage`	`boolean`	false	When true, indicates that the user wants to copy the application in the server staging area externally or using a third-party tool. When specified, WebLogic Server looks for the application under StagingDirectoryName(of target server)/applicationName. Default value is: false
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: true
`id`	`java.lang.String`	false	Specifies an optional, user-supplied, unique deployment task identifier.
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`name`	`java.lang.String`	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.

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

Table 3-7 (Cont.) distribute-app Parameters

Name	Type	Required	Description
nostage	boolean	false	When true, does not copy the deployment files to target servers, but leaves them in a fixed location, specified by the source parameter. By default, nostage is true for the Administration Server and stage is true for the Managed Server targets.
noversion	boolean	false	When true, ignores all version related code paths on the Administration Server. Default value is: false
nowait	boolean	false	When true, initiates multiple tasks and then monitors them later with the -list action. Default value is: false
password	java.lang.String	false	Specifies the administrative password.
plan	java.lang.String	false	Specifies the path to the deployment plan.
remote	boolean	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the source parameter specifies a path on the server, unless the upload parameter is also used. Default value is: false
retiretimeout	java.lang.Integer	false	Specifies the number of seconds before WebLogic Server undeploys the currently-running version of this application or module so that clients can start using a new version. When not specified, a graceful retirement policy is assumed. Default value is: -1
securityModel	java.lang.String	false	Specifies the security model to be used for this deployment, overriding the default security model for the security realm. Possible values are: DDOnly, CustomRoles, CustomRolesAndPolicies, and Advanced.
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
source	java.lang.String	false	Specifies the address of the artifact to distribute. The address can be one of the following: • A colon (:) separated list of Maven coordinates of the form: groupId:artifactId:packaging:classifier:version. • An archive file or exploded archive directory on the local system. For example, /home/myhome/myapps/helloworld.war. • A remote HTTP URL (http://foo/a/b.ear).
stage	boolean	false	When true, indicates that the application needs to be copied into the target server staging area before deployment. By default, nostage is true for the Administration Server and stage is true for the Managed Server targets.
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	Specifies a comma-separated list of targets for the current operation. When not specified, all configured targets are used. For a new application, the default target is the Administration Server.

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

Table 3-7 (Cont.) distribute-app Parameters

Name	Type	Required	Description
`timeout`	`java.lang.Integer`	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1
`upload`	`boolean`	false	When true, copies the source files to the Administration Server's upload directory prior to deployment. Use this setting when running the plug-in remotely (using the remote parameter) and when the user lacks normal access to the Administration Server's file system. Default value is: false
`user`	`java.lang.String`	false	Specifies the administrative user name.
`userConfigFile`	`java.lang.String`	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
`userKeyFile`	`java.lang.String`	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
`verbose`	`boolean`	false	When true, displays additional status information. Default value is: false
`version`	`boolean`	false	When true, prints the version information. Default value is: false
`weblogicHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.

Use this goal to prepare deployment files for deployment.

```
<execution>
<id>wls-distribute-app</id>
<phase>pre-integration-test</phase>
<goals>
<goal>distribute-app</goal>
</goals>
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<source>${project.build.directory}/${project.build.finalName}
.${project.packaging}</source>
<targets>cluster1</targets>
<verbose>true</verbose>
<name>${project.build.finalName}</name>
</configuration>
</execution>
```

Example 3-6 shows typical `distribute-app` goal output.

Example 3-6 distribute-app

```
$ mvn com.oracle.weblogic:weblogic-maven-plugin:distribute-app
 -Dadminurl=t3://localhost:7001 -Dstage=true -DmiddlewareHome=/maven/wls261000
```


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

```
 -Dname=cluster-test -Duser=weblogic -Dpassword=password -Dtargets=cluster1
 -Dsource=target/cluster-test-1.0-SNAPSHOT.war
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building cluster-test 1.0-SNAPSHOT
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:distribute-app (default-cli) @
 cluster-test ---
weblogic.Deployer invoked with options: -noexit -adminurl t3://localhost:7001
 -distribute -user weblogic -name cluster-test -source
/home/oracle/src/tests/uber-test/cluster-test/
target/cluster-test-1.0-SNAPSHOT.war -targets cluster1 -stage
<Aug 19, 2015> <Info> <J2EE Deployment SPI> <BEA-260121>
 <Initiating distribute operation for application, cluster-test [archive:
/home/oracle/src/tests/uber-test/cluster-test/
target/cluster-test-1.0-SNAPSHOT.war], to cluster1 .>
Task 0 initiated: [Deployer:149026]distribute application cluster-test on
 cluster1.
Task 0 completed: [Deployer:149026]distribute application cluster-test on
 cluster1.
Target state: distribute completed on Cluster cluster1

[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 6.953s
[INFO] Finished at: Wed Aug 19 14:10:00 EST 2015
[INFO] Final Memory: 15M/429M
[INFO]
------------------------------------------------------------------------
```

install

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:install`

Description

Installs WebLogic Server from a JAR file.

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

Parameters

Table 3-8 install Parameters

Name	Type	Required	Description
`artifactLocation`	`java.lang.String`	true	Specifies the address of the installation. The address can be one of the following: • A colon (:) separated list of Maven coordinates of the form: groupId:artifactId:packaging:classifier:version. • A file on the local system (/home/myhome/myapps/wls_generic.jar). • A remote HTTP URL (http://myarchive/installers/wls_generic.jar).
`installCommand`	`java.lang.String`	false	Installs the product with a binary or jar installer (including the quickstart installers.) The following macros are supported: • @INSTALLER_FILE@ - the path to the installer file. • @INSTALL_TO_LOCATION@ - the target directory (only relevant for the quickstart installer). • @JAVA_HOME@ - path to the Java home. • @JAVA_TMPDIR@ - path to the Java temporary directory. • @RESPONSE_FILE@ - path to the OUI silent installer response file. • @INV_PTR_LOC_FILE@ - path to the OUI invPtrLoc file. JAR installer example: @JAVA_HOME@/bin/java -Xms512m -Xmx1024m -Djava.io.tmpdir=@JAVA_TMPDIR@ -jar @INSTALLER_FILE@ -silent -responseFile @RESPONSE_FILE@ -invPtrLoc @INV_PTR_LOC_FILE@ Quick Start JAR installer example: @JAVA_HOME@/bin/java -Xms512m -Xmx1024m -Djava.io.tmpdir=@JAVA_TMPDIR@ -jar @INSTALLER_FILE@ ORACLE_HOME=@INSTALL_TO_LOCATION@ This parameter is optional. If specified for a quickstart installer when the supplementalQuickStartLocation parameter is supplied, the same command is used for the supplemental quickstart installer by replacing the @INSTALLER_FILE@ macro with the file location derived from the supplementalQuickStartLocation parameter. If the @INSTALLER_FILE@ macro is not being used, the install goal replaces the argument following the '-jar' argument in the installCommand string with the supplemental quickstart installer JAR file name.
`invPtrLoc`	`java.lang.String`	false	The silent installer inventory location file. This is required on Unix-based platforms when using the binary or JAR installers.
`middlewareHome`	`java.lang.String`	false	The ORACLE_HOME directory to install into when using the quickstart installer.
`quickStartInstaller`	`boolean`	false	Selects a Quick Start JAR installer. Set artifactLocation and middlewareHome; middlewareHome is the target ORACLE_HOME. installCommand and supplementalQuickStartLocation can also apply. Default: false.
`responseFile`	`java.lang.String`	false	The silent installer response file. This is required when using the binary or jar installers.

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

Table 3-8 (Cont.) install Parameters

Name	Type	Required	Description
`supplementalQuickStartLocation`	`java.lang.String`	false	The Quick Start supplemental installer.

Usage Example

The install goal installs WebLogic Server software from the specified distribution. It does not create a domain or a Managed Server. After installation, use create-domain or WLST to create a domain, then use start-server to start its Administration Server. create-domain; start-server

This goal installs WebLogic Server using a specified installation distribution. You specify the location of the distribution using the `artifactLocation` configuration parameter, which can be the location of the distribution as a file on the file system; an HTTP URL which can be accessed; or a Maven coordinate of the distribution installed in a Maven repository. Specify the `artifactLocation` configuration element in the `weblogic-maven-plugin` section of the `pom.xml` file, or by using the `-DartifactLocation` property when invoking Maven.

Example 3-7 shows an example of installing WebLogic Server using a JAR file on a Windows-based system.

Example 3-7 Install From JAR File

Prepare response.txt for the selected installer; that response file supplies its ORACLE_HOME.

```
mvn com.oracle.weblogic:weblogic-maven-plugin:26.1.0-0-0:install -
DartifactLocation=c:\wls-temp\wls_jrf_generic.jar -DresponseFile=c:\wls-
temp\response.txt
```

Example 3-8 shows an example of installing WebLogic Server using a JAR file and the `installCommand` parameter on a Windows-based system.

Example 3-8 Install From JAR File With installCommand

```
mvn com.oracle.weblogic:weblogic-maven-plugin:install -
DinstallCommand="@JAVA_HOME@/bin/java -Xms512m -Xmx1024m -jar
@INSTALLER_FILE@ -silent -responseFile c:\wls-temp\response.txt" -
DartifactLocation=c:\wls-temp\wls_jrf_generic.jar -DresponseFile=c:\wls-
temp\response.txt
```

list-apps

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:list-apps`

Description

Lists the deployment names for applications and standalone modules deployed, distributed, or installed in the domain. Does not require a local server installation.

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

Parameters

Table 3-9 list-apps Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`advanced`	`boolean`	false	When true, prints advanced usage options.
`debug`	`boolean`	false	When true, displays debug-level messages to the standard output. Default value is: `false`
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: `true`
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`noversion`	`boolean`	false	When true, ignore all version-related code paths on the Administration Server. Default value is: `false`
`nowait`	`boolean`	false	When true, initiates multiple tasks and then monitors them later with the `-list` action.
`password`	`java.lang.String`	false	Specifies the administrative password.
`remote`	`boolean`	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the `source` parameter specifies a path on the server, unless the `upload` parameter is also used.
`serverClasspath`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`timeout`	`java.lang.Integer`	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: `-1`
`user`	`java.lang.String`	false	Specifies the administrative user name.
`userConfigFile`	`java.lang.String`	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
`userKeyFile`	`java.lang.String`	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
`verbose`	`boolean`	false	When true, displays additional status information. Default value is: `false`
`version`	`boolean`	false	When true, prints the version information. Default value is: `false`
`weblogicHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.

Use the list-apps goal to list the deployment names.

```
<execution>
<id>wls-list-apps</id>
<phase>pre-integration-test</phase>
<goals>
<goal>list-apps</goal>
</goals>
```


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

```
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
</configuration>
</execution>
```

Example 3-9 shows typical `list-apps` goal output.

Example 3-9 list-apps

```
mvn com.oracle.weblogic:weblogic-maven-plugin:list-apps
-Duser=weblogic -Dpassword=password
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 15.1.1.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:15.1.1-0-0:list-apps (default-cli) @ weblogic-m
aven-plugin ---
weblogic.Deployer invoked with options:  -noexit -adminurl t3://localhost:7001 -
listapps -user weblogic
 SamplesSearchWebApp
 stockBackEnd
 ajaxJSF
 asyncServlet30
 singletonBean
 webFragment
 examplesWebApp
 mainWebApp
 annotation
 MySimpleEjb
 stockFrontEnd
 jsfBeanValidation
 programmaticSecurity
 entityBeanValidation
 faceletsJSF
 bookmarkingJSF
 stockAdapter
 noInterfaceViewInWAR
 jdbcDataSource.war
 asyncMethodOfEJB
 calendarStyledTimer
 cdi
 jaxrs
 criteriaQuery
 portableGlobalJNDIName
 multipartFileHandling
 elementCollection
Number of Applications Found : 27
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 8.656s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 11M/28M
[INFO] ------------------------------------------------------------------------
C:\Oracle\Middleware\Oracle_Home\wlserver\server\lib>
```


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

purge-tasks

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:purge-tasks`

Description

Flushes out retired deployment tasks.

Parameters

Table 3-10 purge-tasks Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`debug`	`boolean`	false	When true, compiles debugging information into class files. Default value is: `false`
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: `true`
`password`	`java.lang.String`	false	Specifies the administrative password.
`user`	`java.lang.String`	false	Specifies the administrative user name.
`userConfigFile`	`java.lang.String`	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
`userKeyFile`	`java.lang.String`	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
`verbose`	`boolean`	false	When true, displays additional status information during the deployment process. Default value is: `false`

Use the purge-tasks goal to flush out retired deployment tasks.

```
<execution>
<id>wls-purge</id>
<phase>pre-integration-test</phase>
<goals>
<goal>purge-tasks</goal>
</goals>
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
</configuration>
</execution>
```

Example 3-11 shows typical `purge-tasks` goal output.

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

Example 3-10 purge-tasks

```
mvn  com.oracle.weblogic:weblogic-maven-plugin:purge-task
s -Duser=weblogic -Dpassword=password
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:15.1.1-0-0:purge-tasks (default-cli) @ standalo
ne-pom ---
weblogic.Deployer invoked with options:  -noexit -purgetasks -user weblogic -adm
inurl t3://localhost:7001
Currently there are no retired tasks.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 13.139s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 8M/24M
[INFO] ------------------------------------------------------------------------
```

redeploy

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:redeploy`

Description

Redeploys a running application or part of a running application. Does not require a local server installation.

Parameters

Table 3-11 redeploy Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`appversion`	`java.lang.String`	false	Version of the application to start.
`deleteFiles`	`java.lang.String`	false	Removes the files specified in this parameter while leaving the application activated. This parameter is valid only for unarchived deployments.
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: `true`
`id`	`java.lang.String`	false	Specifies an optional, user-supplied, unique deployment task identifier.
`libimplver`	`java.lang.String`	false	Implementation version of a Jakarta EE library or optional package. This option can be used only if the library or package does not include an implementation version in its manifest file.
`library`	boolean	false	Deploy as a shared Jakarta EE library or optional package.

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

Table 3-11 (Cont.) redeploy Parameters

Name	Type	Required	Description
libspecver	java.lang.String	false	Specification version of a Jakarta EE library or optional package. This option can be used only if the library or package does not include a specification version in its manifest file.
middlewareHome	java.lang.String	false	This parameter is deprecated in this release and ignored.
name	java.lang.String	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.
password	java.lang.String	false	Specifies the administrative password.
plan	java.lang.String	false	Specifies the path to the deployment plan.
remote	boolean	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the source parameter specifies a path on the server, unless the upload parameter is also used.
removePlanOverride	boolean	false	Removes an overridden deployment plan during a redeploy or update deployment action. To remove an application override, specify the removePlanOverride attribute.
retiretimeout	java.lang.Integer	false	Specifies the number of seconds before WebLogic Server undeploys the currently running version of this application or module so that clients can start using a new version. When not specified, a graceful retirement policy is assumed. Default value is: -1
rmiGracePeriod	java.lang.Integer	false	Specifies the number of seconds in the grace period for RMI requests during graceful shutdown. Can be used only when the graceful parameter is true. The default value of -1 means no grace period. Default value is: -1
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
source	java.lang.String	false	Specifies the address of the artifact to redeploy. The address can be one of the following: • A colon (:) separated list of Maven coordinates of the form: groupId:artifactId:packaging:classifier:version. • An archive file or exploded archive directory on the local system. For example, /home/myhome/myapps/helloworld.war. • A remote HTTP URL (http://foo/a/b.ear).
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	Specifies a comma-separated list of targets for the current operation. The default target is AdminServer.
timeout	java.lang.Integer	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1

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

Table 3-11 (Cont.) redeploy Parameters

Name	Type	Required	Description
upload	boolean	false	When true, copies the specified source files to the Administration Server's upload directory prior to redeployment. Use this setting when running the plug-in remotely (using the remote parameter) and when the user lacks normal access to the Administration Server's file system. Default value is: false
user	java.lang.String	false	Specifies the administrative user name.
userConfigFile	java.lang.String	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
userKeyFile	java.lang.String	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
verbose	boolean	false	When true, displays additional status information during the deployment process. Default value is: false
version	boolean	false	When true, prints the version information. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

Use the redeploy goal to redeploy an application or part of that application.

```
<execution>
<id>wls-redeploy</id>
<phase>pre-integration-test</phase>
<goals>
<goal>redeploy</goal>
</goals>
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<source>${project.build.directory}/${project.build.finalName}.${project.packaging}</sour
ce>
<name>${project.build.finalName}</name>
</configuration>
</execution>
```

Example 3-11 shows typical redeploy goal output.

Example 3-11 redeploy

```
mvn com.oracle.weblogic:weblogic-maven-plugin:redeploy -Dsou
rce=C:\Oracle\Middleware\Oracle_Home\wlserver\server\lib\MySimpleEjb.jar  -Duser
=weblogic -Dpassword=password -Dname=ExampleEJB
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:redeploy (default-cli) @ weblogic-ma
ven-plugin ---
weblogic.Deployer invoked with options:  -noexit -adminurl t3://localhost:7001 -
```


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

```
redeploy -user weblogic -name ExampleEJB -source C:\Oracle\Middleware\Oracle_Hom
e\wlserver\server\lib\MySimpleEjb.jar -targets AdminServer
<Aug 19, 2015> <Info> <J2EE Deployment SPI> <BEA-260121> <Initiat
ing redeploy operation for application, ExampleEJB [archive: C:\Oracle\Middlewar
e\Oracle_Home\wlserver\server\lib\MySimpleEjb.jar], to AdminServer .>
Task 3 initiated: [Deployer:149026]deploy application ExampleEJB on AdminServer.

Task 3 completed: [Deployer:149026]deploy application ExampleEJB on AdminServer.

Target state: redeploy completed on Server AdminServer

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6.322s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
```

remove-domain

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:remove-domain`

Description

Removes a domain directory. The domain must not be running for this goal to succeed. This is a convenience goal for the simple use case. If the domain is already removed, stdout prints a status message but the goal does not fail.

Parameters

Table 3-12 remove-domain Parameters

Name	Type	Required	Description
`domainHome`	`java.lang.String`	true	The path to the domain directory.
`workingDir`	`java.lang.String`	false	Specifies the current working directory. Default value is: `${project.build.directory}/weblogic-maven-plugin)`

Use the remove-domain goal to remove a domain directory.

```
<execution>
<id>wls-remove-domain</id>
<phase>pre-integration-test</phase>
<goals>
<goal>remove-domain</goal>
</goals>
<configuration>
<domainHome>${project.build.directory}/base_domain</domainHome>
</configuration>
</execution>
```

Example 3-13 shows typical `remove-domain` goal output.

Example 3-12 remove-domain

```
mvn com.oracle.weblogic:weblogic-maven-plugin:remove-domain
-DdomainHome=C:\Oracle\Middleware\Oracle_Home\user_projects\domains\base_domain
```


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

```
:
[INFO] [remove-domain]Executing: [cmd:[C:\\Windows\\System32\\cmd.exe, /c, rmdir
 /Q /S C:\Oracle\Middleware\Oracle_Home\user_projects\domains\base_domain]]
[INFO] Process being executed, waiting for completion.
[INFO] [remove-domain][cmd:[C:\\Windows\\System32\\cmd.exe, /c, rmdir /Q /S C:\O
racle\Middleware\Oracle_Home\user_projects\domains\base_domain]] exit code=0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4:01.074s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 8M/20M
[INFO] ------------------------------------------------------------------------
```

start-app

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:start-app`

Description

Starts an application deployed on WebLogic Server. Does not require a local server installation.

Parameters

Table 3-13 start-app Parameters

Name	Type	Required	Description
`adminmode`	`boolean`	false	When true, switches the application to administration mode so that it accepts only administration requests via a configured administration channel. When false, production mode is assumed. Default value is: `false`
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`advanced`	`boolean`	false	When true, prints advanced usage options.
`appversion`	`java.lang.String`	false	Specifies the version identifier of the application. When not specified, the currently active version of the application is assumed.
`debug`	`boolean`	false	When true, displays debug-level messages to the standard output. Default value is: `false`
`domainHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: `true`
`id`	`java.lang.String`	false	Specifies an optional, user-supplied, unique deployment task identifier.
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`name`	`java.lang.String`	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.

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

Table 3-13 (Cont.) start-app Parameters

Name	Type	Required	Description
noversion	boolean	false	When true, ignores all version-related code paths on the Administration Server. Default value is: false
nowait	boolean	false	When true, initiates multiple tasks and then monitors them later with the -list action.
password	java.lang.String	false	Specifies the administrative password.
planversion	java.lang.String	false	Specifies the version of the deployment plan. When not specified, the currently active version of the application's deployment plan is assumed.
remote	boolean	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the source parameter specifies a path on the server, unless the upload parameter is also used. Default value is: false
retiretimeout	java.lang.Integer	false	Specifies the number of seconds before WebLogic Server undeploys the currently running version of this application or module so that clients can start using a new version. When not specified, a graceful retirement policy is assumed. Default value is: -1
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	Specifies a comma-separated list of targets for the current operation. When not specified, all configured targets are used. For a new application, the default target is all targets to which the application is deployed.
timeout	java.lang.Integer	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1
user	java.lang.String	false	Specifies the administrative user name.
userConfigFile	java.lang.String	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
userKeyFile	java.lang.String	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
verbose	boolean	false	When true, displays additional status information during the deployment process. Default value is: false
version	boolean	false	When true, prints the version information. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

Use the start-app goal to start an application.

```
<execution>
<id>wls-start-app</id>
```


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

```
<phase>pre-integration-test</phase>
<goals>
<goal>start-app</goal>
</goals>
<configuration>
<adminurl>t3://localhost:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<name>${project.build.finalName}</name>
</configuration>
</execution>
```

Example 3-13 shows typical `start-app` goal output.

Example 3-13 start-app

```
mvn com.oracle.weblogic:weblogic-maven-plugin:start-app
-Duser=weblogic -Dpassword=password -Dname=ExampleEJB
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:start-app (default-cli) @ weblogic-m
aven-plugin ---
weblogic.Deployer invoked with options:  -noexit -adminurl t3://localhost:7001 -
start -user weblogic -name ExampleEJB -retiretimeout -1
<Aug 19, 2015> <Info> <J2EE Deployment SPI> <BEA-260121> <Initiat
ing start operation for application, ExampleEJB [archive: null], to configured t
argets.>
Task 5 initiated: [Deployer:149026]start application ExampleEJB on AdminServer.
Task 5 completed: [Deployer:149026]start application ExampleEJB on AdminServer.
Target state: start completed on Server AdminServer

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6.053s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 10M/26M
[INFO] ------------------------------------------------------------------------
```

start-server

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:start-server`

Description

Starts WebLogic Server from a script in the current working directory. This is a convenience goal for the simple use case. If the server is already started, stdout prints a status message but the goal does not fail.

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

Parameters

Table 3-14 start-server Parameters

Name	Type	Required	Description
command	java.lang.String[]	false	Specifies the script to start WebLogic Server. If this parameter is not specified, it will default to either startWebLogic.sh or startWebLogic.cmd, based on the platform.
domainHome	java.lang.String	false	Specifies the path to the WebLogic Server domain. Default value is: ${basedir}/Oracle/Domains/mydomain
httpPingUrl	java.lang.String	false	Specifies the URL that, when pinged, will verify that the server is running.
middlewareHome	java.lang.String	false	This parameter is deprecated in this release and ignored.
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
timeoutSecs	java.lang.Integer	false	Specifies in seconds, the timeout for the script. Valid when the waitForExit parameter is true. A zero (0) or negative value indicates that the script will not timeout. Default value is: -1
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

Usage Example

The start-server goal executes a startWebLogic command on a given domain, starting the WebLogic Server instance.

```
<execution>
<id>wls-wlst-start-server</id>
<phase>pre-integration-test</phase>
<goals>
<goal>start-server</goal>
</goals>
<configuration>
<domainHome>${project.build.directory}/base_domain</domainHome>
</configuration>
</execution>
```

Example 3-14 shows typical start-server goal output.

Example 3-14 start-server

```
mvn com.oracle.weblogic:weblogic-maven-plugin:start-server
-DdomainHome=c:\oracle\middleware\oracle_home\user_projects\domains\wl_server
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0-0-0
[INFO] ------------------------------------------------------------------------[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:start-server (default-cli)
@ weblogic-maven-plugin ---
.[INFO] Starting server in domain:
c:\oracle\middleware\oracle_home\user_projects\domains\wl_server
[INFO] Check stdout file for details:
c:\oracle\middleware\oracle_home\user_projects\domains\wl_server\server-21831141069721263
86.out
```


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

```
[INFO] Process being executed, waiting for completion.
.............
[INFO] Server started successful
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 37.725s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 8M/23M
```

stop-app

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:stop-app`

Description

Stops an application. Does not require a local server installation.

Parameters

Table 3-15 stop-app Parameters

Name	Type	Required	Description
`adminmode`	`boolean`	false	When true, switches the application to administration mode so that it accepts only administration requests via a configured administration channel. When false, production mode is assumed. Default value is: `false`
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`advanced`	`boolean`	false	When true, prints advanced usage options.
`appversion`	`java.lang.String`	false	Specifies the version identifier of the application. When not specified, the currently active version of the application is assumed.
`debug`	`boolean`	false	When true, displays debug-level messages to the standard output. Default value is: `false`
`domainHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: `true`
`graceful`	`boolean`	false	When true, stops the application after existing HTTP clients have completed their work. When not specified, force shutdown is assumed.
`id`	`java.lang.String`	false	Specifies an optional, user-supplied, unique deployment task identifier.
`ignoresessions`	`boolean`	false	When true, ignores pending HTTP sessions during graceful shutdown. Can be used only when the `graceful` parameter is `true`. Default value is: `false`
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.

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

Table 3-15 (Cont.) stop-app Parameters

Name	Type	Required	Description
name	java.lang.String	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.
noversion	boolean	false	When true, ignores all version-related code paths on the Administration Server. Default value is: false
nowait	boolean	false	When true, initiates multiple tasks and then monitors them later with the -list action.
password	java.lang.String	false	Specifies the administrative password.
planversion	java.lang.String	false	Specifies the version of the deployment plan. When not specified, the currently active version of the application's deployment plan is assumed.
remote	boolean	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the source parameter specifies a path on the server, unless the upload parameter is also used. Default value is: false
rmiGracePeriod	java.lang.Integer	false	Specifies the number of seconds in the grace period for RMI requests during graceful shutdown. Can be used only when the graceful parameter is true. The default value of -1 means no grace period. Default value is: -1
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	Specifies a comma-separated list of targets for the current operation. When not specified, all configured targets are used.
timeout	java.lang.Integer	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1
user	java.lang.String	false	Specifies the administrative user name.
userConfigFile	java.lang.String	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
userKeyFile	java.lang.String	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
verbose	boolean	false	When true, displays additional status information. Default value is: false
version	boolean	false	When true, prints the version information. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

Use the stop-app goal to stop an application.

```
<execution>
<id>wls-start-app</id>
<phase>pre-integration-test</phase>
<goals>
<goal>start-app</goal>
</goals>
```


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

```
<configuration>
<adminurl>t3://localhost:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<name>${project.build.finalName}</name>
</configuration>
</execution>
```

Example 3-15 shows typical `stop-app` goal output.

Example 3-15 stop-app

```
mvn com.oracle.weblogic:weblogic-maven-plugin:stop-app  -Dus
er=weblogic -Dpassword=password -Dname=ExampleEJB
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:stop-app (default-cli)
@ weblogic-ma
ven-plugin ---
weblogic.Deployer invoked with options:  -noexit
-adminurl t3://localhost:7001 -
stop -user weblogic -name ExampleEJB
<Aug 19, 2015> <Info>
<J2EE Deployment SPI> <BEA-260121> <Initiat
ing stop operation for application, ExampleEJB [archive: null],
to configured ta
rgets.>
Task 6 initiated: [Deployer:149026]stop application ExampleEJB on
AdminServer.
Task 6 completed: [Deployer:149026]stop application ExampleEJB on
AdminServer.
Target state: stop completed on Server AdminServer

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6.028s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 10M/29M
[INFO] ------------------------------------------------------------------------
C:\Oracle\Middleware\Oracle_Home\wlserver\server\lib>
```

stop-server

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:stop-server`

Description

Stops WebLogic Server from a script in the current working directory. This is a convenience goal for the simple use case. If the server is already stopped, stdout prints a status message but the goal does not fail.

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

Parameters

Table 3-16 stop-server Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`command`	`java.lang.String[]`	false	Specifies the script to stop WebLogic Server. This will default to `stopWebLogic.sh` or `stopWebLogic.cmd`, based on the platform.
`domainHome`	`java.lang.String`	false	Specifies the path to the WebLogic Server domain. Default value is: `${basedir}/Oracle/Domains/mydomain`
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`outputLog`	`java.lang.String`	false	Specifies the log file to which the script output will be redirected. When not specified, it defaults to `stdout`.
`password`	`java.lang.String`	**true**	Specifies the administrative password.
`timeoutSecs`	`java.lang.Integer`	false	Specifies, in seconds, the timeout for the script. This is valid when the `waitForExit` parameter is `true`. A zero (`0`) or negative value indicates that the script will not timeout. Default value is: `-1`
`user`	`java.lang.String`	**true**	Specifies the administrative user name.
`waitForExit`	`boolean`	false	When true, the plug-in should wait for the script to complete. Default value is: `true`
`weblogicHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`workingDir`	`java.lang.String`	false	Specifies the working directory for the script. If you do not specify this attribute, it defaults to the current working directory. Default value is: `${project.base.directory}`

Usage Example

The `stop-server` goal stops a server instance using the `stopWebLogic` script in the specified domain.

```
<execution>
<id>wls-wlst-stop-server</id>
<phase>post-integration-test</phase>
<goals>
<goal>stop-server</goal>
</goals>
<configuration>
<domainHome>${project.build.directory}/base_domain</domainHome>
<user>weblogic</user>
<password>password</password>
<adminurl>t3://localhost:7001</adminurl>
</configuration>
</execution>
```

Example 3-16 shows typical `stop-server` goal output.

Example 3-16 stop-server

```
mvn com.oracle.weblogic:weblogic-maven-plugin:stop-server
 -DdomainHome=c:\oracle\middleware\oracle_home\userprojects\domains\wl_server
-DworkingDir=c:\oracle\middleware\oracle_home\user_projects\domains\wl_server
```


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

```
-Duser=weblogic -Dpassword=password
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0-0-0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:stop-server (default-cli)
@ weblogic
-maven-plugin ---
[INFO] Stop server in domain:
c:\oracle\middleware\oracle_home\user_projects\dom
ains\wl_server
[INFO] Process being executed, waiting for completion.
[INFO] [exec] Stopping Weblogic Server...
[INFO] [exec]
[INFO] [exec] Initializing WebLogic Scripting Tool (WLST) ...
[INFO] [exec]
[INFO] [exec] Welcome to WebLogic Server Administration Scripting Shell
[INFO] [exec]
[INFO] [exec] Type help() for help on available commands
[INFO] [exec]
[INFO] [exec] Connecting to t3://localhost:7001 with userid weblogic ...
[INFO] [exec] Successfully connected to Admin Server "AdminServer" that belongs
to domain "wl_server".
[INFO] [exec]
[INFO] [exec] Warning: An insecure protocol was used to connect to the
[INFO] [exec] server. To ensure on-the-wire security, the SSL port or
[INFO] [exec] Admin port should be used instead.
[INFO] [exec]
[INFO] [exec] Shutting down the server AdminServer with force=false while connec
ted to AdminServer ...
[INFO] [exec] WLST lost connection to the WebLogic Server that you were
[INFO] [exec] connected to, this may happen if the server was shutdown or
[INFO] [exec] partitioned. You will have to re-connect to the server once the
[INFO] [exec] server is available.
[INFO] [exec] Disconnected from weblogic server: AdminServer
[INFO] [exec] Disconnected from weblogic server:
[INFO] [exec]
[INFO] [exec]
[INFO] [exec] Exiting WebLogic Scripting Tool.
[INFO] [exec]
[INFO] [exec] Done
[INFO] [exec] Stopping Derby Server...
[INFO] [exec] Derby server stopped.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 23.270s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 9M/23M
[INFO] ------------------------------------------------------------------------
```

undeploy

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:undeploy`

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

Description

Undeploys the application from WebLogic Server. Stops the deployment unit and removes staged files from target servers. Does not require a local server installation.

Parameters

Table 3-17 undeploy Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`advanced`	`boolean`	false	When true, prints advanced usage options.
`appversion`	`java.lang.String`	false	Specifies the version identifier of the application. When not specified, the currently active version of the application is assumed.
`debug`	`boolean`	false	When true, displays debug-level messages to the standard output. Default value is: `false`
`examples`	`boolean`	false	When true, displays examples of how to use this plug-in.
`failOnError`	`boolean`	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: `true`
`graceful`	`boolean`	false	When true, stops the application after existing HTTP clients have completed their work. When not specified, forced shutdown is assumed.
`id`	`java.lang.String`	false	Specifies an optional, user-supplied, unique deployment task identifier.
`ignoresessions`	`boolean`	false	When true, ignores pending HTTP sessions during graceful shutdown. Can be used only when the `graceful` parameter is `true`. Default value is: `false`
`middlewareHome`	`java.lang.String`	false	This parameter is deprecated in this release and ignored.
`name`	`java.lang.String`	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.
`noversion`	`boolean`	false	When true, ignores all version-related code paths on the Administration Server. Default value is: `false`
`nowait`	`boolean`	false	When true, initiates multiple tasks and then monitors them later with the `-list` action.
`password`	`java.lang.String`	false	Specifies the administrative password.
`planversion`	`java.lang.String`	false	Specifies the version of the deployment plan. When not specified, the currently active version of the application's deployment plan is assumed.
`remote`	`boolean`	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the `source` parameter specifies a path on the server, unless the `upload` parameter is also used. Default value is: `false`
`rmiGracePeriod`	`java.lang.Integer`	false	Specifies the number of seconds in the grace period for RMI requests during graceful shutdown. Can be used only when the `graceful` parameter is `true`. The default value of `-1` means no grace period. Default value is: `-1`

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

Table 3-17 (Cont.) undeploy Parameters

Name	Type	Required	Description
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	Specifies a comma-separated list of targets for the current operation. When not specified, all configured targets are used.
timeout	java.lang.Integer	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1
user	java.lang.String	false	Specifies the administrative user name.
userConfigFile	java.lang.String	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
userKeyFile	java.lang.String	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
verbose	boolean	false	When true, displays additional status information during the deployment process. Default value is: false
version	boolean	false	When true, prints the version information. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

Use the undeploy goal to undeploy an application from WebLogic Server.

```
<execution>
<id>wls-undeploy</id>
<phase>post-integration-test</phase>
<goals>
<goal>undeploy</goal>
</goals>
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<name>${project.build.finalName}</name>
</configuration>
</execution>
```

Example 3-17 shows typical undeploy goal output.

Example 3-17 undeploy

```
mvn com.oracle.weblogic:weblogic-maven-plugin:undeploy
-Duser=weblogic -Dpassword=password -Dname=ExampleEJB
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building WebLogic Server Maven Plugin 26.1.0.0.0
[INFO] ------------------------------------------------------------------------
```


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

```
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:undeploy (default-cli)
@ weblogic-ma
ven-plugin ---
weblogic.Deployer invoked with options:  -noexit
-adminurl t3://localhost:7001 -
undeploy -user weblogic -name ExampleEJB -targets AdminServer
<Aug 19, 2015> <Info> <J2EE Deployment SPI>
<BEA-260121> <Initiat
ing undeploy operation for application, ExampleEJB [archive: null],
to AdminServ
er .>
Task 7 initiated: [Deployer:149026]remove application ExampleEJB
on AdminServer.

Task 7 completed: [Deployer:149026]remove application ExampleEJB
on AdminServer.

Target state: undeploy completed on Server AdminServer

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6.114s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 9M/26M
[INFO] ------------------------------------------------------------------------
```

uninstall

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:uninstall`

Description

Uninstalls WebLogic Server.

Parameters

Table 3-18 uninstall Parameters

Name	Type	Required	Description
`invPtrLoc`	`java.io.File`	true	This parameter is deprecated and ignored.
`middlewareHome`	`java.lang.String`	true	The Oracle Middleware installation directory. This parameter is required when uninstalling a server installed using the Quickstart installer. Otherwise, it is ignored and the location in the responseFile is used.
`response`	`java.io.File`	true	Deprecated. Use the responseFile parameter.
`responseFile`	`java.io.File`	true	The silent installer response file. This is required when using the binary or JAR installers.

Example 3-18 shows an example of uninstalling WebLogic Server in a JAR file installation.

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

Example 3-18 uninstall in JAR Installation

```
mvn com.oracle.weblogic:weblogic-maven-plugin:uninstall -DresponseFile=c:\wls-
temp\response.txt
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:uninstall (default-cli) @
standalone
-pom ---
[INFO] [uninstall]ORACLE_HOME = C:\test-maven\Oracle\Middleware\Oracle_Home
[INFO] [uninstall]ORACLE_HOME = C:\test-maven\Oracle\Middleware\Oracle_Home
[INFO] Executing:  [cmd:[C:\\Windows\\System32\\cmd.exe, /c, C:\test-
maven\Oracl
e\Middleware\Oracle_Home\oui\bin\deinstall.cmd -noconsole -deinstall -silent -
re
sponseFile c:\wls-temp\response.txt]]
[INFO] Process being executed, waiting for completion.
[INFO] Installer exited with code: 0
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
```

update-app

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:update-app`

Description

Updates an application's deployment plan by redistributing the plan files and reconfiguring the application based on the new plan contexts. Does not require a local server installation.

Parameters

Table 3-19 update-app Parameters

Name	Type	Required	Description
`adminurl`	`java.lang.String`	false	Specifies the listen address and listen port of the Administration Server. Default value is: `t3://localhost:7001`
`advanced`	`boolean`	false	When true, prints advanced usage options.
`appversion`	`java.lang.String`	false	Specifies the version identifier of the application. When not specified, the currently active version of the application is assumed.

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

Table 3-19 (Cont.) update-app Parameters

Name	Type	Required	Description
debug	boolean	false	When true, displays debug-level messages to the standard output. Default value is: false
domainHome	java.lang.String	false	This parameter is deprecated in this release and ignored.
examples	boolean	false	When true, displays examples of how to use this plug-in.
failOnError	boolean	false	When true, forces the Mojo to fail the build upon encountering an error if it would otherwise just log the error. Default value is: true
id	java.lang.String	false	Specifies an optional, user-supplied, unique deployment task identifier.
middlewareHome	java.lang.String	false	This parameter is deprecated in this release and ignored.
name	java.lang.String	false	Specifies the deployment name to assign to a newly-deployed application or standalone module.
noversion	boolean	false	When true, ignores all version-related code paths on the Administration Server. Default value is: false
nowait	boolean	false	When true, initiates multiple tasks and then monitors them later with the -list action.
password	java.lang.String	false	Specifies the administrative password.
plan	java.lang.String	false	Specifies the location of the deployment plan.
planversion	java.lang.String	false	Specifies the version of the deployment plan. When not specified, the currently active version of the application's deployment plan is assumed.
remote	boolean	false	When true, specifies that the plug-in is not running on the same machine as the Administration Server. In this case, the source parameter specifies a path on the server, unless the upload parameter is also used. Default value is: false
removePlanOverride	boolean	false	Removes an overridden deployment plan during a redeploy or update deployment action. To remove an application override, specify the removePlanOverride attribute.
rmiGracePeriod	java.lang.Integer	false	Specifies the number of seconds in the grace period for RMI requests during graceful shutdown. Can be used only when the graceful parameter is true. The default value of -1 means no grace period. Default value is: -1
serverClasspath	java.lang.String	false	This parameter is deprecated in this release and ignored.
submoduletargets	java.lang.String	false	Specifies JMS Server targets for resources defined within a JMS application module. Possible values have the form: submod@mod-jms.xml@target or submoduleName@target.
targets	java.lang.String	false	The targets on which to update the application or module. This attribute can be a comma-separated list. If no targets are specified, all targets are updated.
timeout	java.lang.Integer	false	Specifies the maximum number of seconds WebLogic Server will wait for the deployment task to complete. The default value of -1 means wait forever. Default value is: -1

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

Table 3-19 (Cont.) update-app Parameters

Name	Type	Required	Description
upload	boolean	false	When true, copies the source files to the Administration Server's upload directory prior to deployment. Use this setting when running the plug-in remotely (using the remote parameter) and when the user lacks normal access to the Administration Server's file system. Default value is: false
user	java.lang.String	false	Specifies the administrative user name.
userConfigFile	java.lang.String	false	Specifies the location of a user configuration file to use for the administrative user name and password instead of specifying the user name and password directly in plain text.
userKeyFile	java.lang.String	false	Specifies the location of a user key file to use for encrypting and decrypting the user name and password stored in the user configuration file.
verbose	boolean	false	When true, displays additional status information. Default value is: false
version	boolean	false	When true, prints the version information. Default value is: false
weblogicHome	java.lang.String	false	This parameter is deprecated in this release and ignored.

Use the update-app goal to update an application's deployment plan.

```
<execution>
<id>wls-update-app</id>
<phase>pre-integration-test</phase>
<goals>
<goal>update-app</goal>
</goals>
<configuration>
<adminurl>t3://127.0.0.1:7001</adminurl>
<user>weblogic</user>
<password>password</password>
<name>${project.build.finalName}</name>
<plan>${basedir}/misc/myplan.xml</plan>
</configuration>
</execution>
```

Example 3-19 shows typical wlst goal output.

Example 3-19 update-app

```
$ mvn com.oracle.weblogic:weblogic-maven-plugin:update-app -Duser=weblogic
 -Dpassword=password -Dadminurl=t3://localhost:7001 -Dplan=misc/myplan.xml
 -Dname=basicWebapp
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building basicWebapp 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:update-app (default-cli)
@ main-test ---
weblogic.Deployer invoked with options: -noexit -adminurl
t3://localhost:7001 -update -user weblogic -plan
 /home/oracle/src/tests/main-test/misc/myplan.xml -name basicWebapp -targets AdminServer
```


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

```
<Aug 19, 2015> <Info> <J2EE Deployment SPI> <BEA-260121>
 <Initiating update operation for application, basicWebapp [archive: null],
 to AdminServer .>
Task 10 initiated: [Deployer:149026]update application basicWebapp on
 AdminServer.
Task 10 completed: [Deployer:149026]update application basicWebapp on
 AdminServer.
Target state: update completed on Server AdminServer

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 10.651s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 18M/435M
[INFO] ------------------------------------------------------------------------
```

wlst

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:wlst`

Description

This goal is a wrapper for the WLST scripting tool. It requires a server install for WLST online commands.

Parameters

Table 3-20 wlst Parameters

Name	Type	Required	Description
`args`	`java.lang.String`	false	Deprecated. Use the scriptArgs parameter to specify the arguments as a list of scriptArg elements. Specifies a string value containing command-line arguments to pass to the WLST Python interpreter. The arguments are delimited by spaces. An argument that contains embedded spaces should be quoted either with single quotes or with escaped double quotes. For example, here is a string for args that contains two parameters: `"'Thomas Paine' \"Now is the time that tries men's souls.\""`
`debug`	`boolean`	false	When true, displays additional status information. Default value is: `false`
`executeScriptBeforeFile`	`boolean`	false	When true, specifies whether a script, if supplied, executes before or after the file, if supplied. Either a file or a script is required, and both are allowed. See `filename` and `script` parameters. Default value is: `true`
`failOnError`	`boolean`	false	When true, the Maven build fails if the `wlst` goal fails. The default value is `true`, and consequently any error condition will cause the build to fail. In some cases, setting `failOnError` to `false` will allow the `wlst` goal to ignore the error. Default value is: `true`

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

Table 3-20 (Cont.) wlst Parameters

Name	Type	Required	Description
`fileName`	`java.lang.String`	false	Specifies the file path of the WLST Python script to execute. Either a `fileName` or a `script` parameter must be specified, and both are allowed.
`middlewareHome`	`java.lang.String`	true	The path to the Oracle Middleware install directory.
`propertiesFile`	`java.lang.String`	false	Specifies the path to a Java properties file. The property names become defined variables in the WLST Python interpreter and are initialized to the values supplied. For example, if the properties file contains the line `"foobar: Very important stuff"`, the variable `foobar` can be used in a Python statement in the following manner: `"print('foobar has the value: ' + foobar)"`.
`script`	`java.lang.String`	false	Specifies an inline WLST Python script, for example, `"print('Hello, world!')"`. Because Python uses indentation to demarcate nested code blocks, scripts that contain multiple lines must be specified in the POM without any indentation within the pom.xml, unless required for code block demarcation.
`scriptArgs`	`java.lang.String`	false	Specifies the command-line arguments to pass to the WLST Jython interpreter as a list of string values. If the argument contains any embedded whitespace, the caller must include enclosing single quotes or escaped double quotes within the scriptArg element's value. If scriptArgs is specified, the args parameter (deprecated) is ignored.
`serverClasspath`	`java.lang.String`	false	This parameter is deprecated and ignored in this release.
`weblogicHome`	`java.lang.String`	false	This parameter is deprecated and ignored in this release.
`wlstVersion`	`java.lang.String`	false	This parameter is deprecated and ignored in this release.
`workingDir`	`java.lang.String`	false	The current working directory where the wlst-script and create-domain goal executes. The default value is: ${project.build.directory}/weblogic-maven-plugin

Usage Example

The `wlst` goal enables the WebLogic Scripting Tool (WLST) to be used to execute scripts that configure resources or perform other operations on a WebLogic Server domain. The `wlst` Maven goal uses the WebLogic Server WLST standard environment so you can use it with all your existing WLST scripts.

You can use the `wlst` goal to execute an external WLST script specified with the `fileName` configuration parameter, or you can specify a sequence of WLST commands within the `pom.xml` file using the `script` configuration element:

```
<execution>
<id>wls-wlst-server</id>
<phase>post-integration-test</phase>
<goals>
<goal>wlst</goal>
</goals>
<configuration>
```


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

```
<middlewareHome>c:/dev/wls14110</middlewareHome>
<fileName>${project.basedir}/misc/configure_resources.py</fileName>
<args>t3://localhost:7001 weblogic password AdminServer</args>
<script>
print('This is a WLST inline script\n')
print('Next, we run a WLST script to create JMS resources on the server\n')
</script>
<executeScriptBeforeFile>true</executeScriptBeforeFile>
</configuration>
</execution>
```

Example 3-20 shows typical `wlst` goal output.

Example 3-20 wlst

```
mvn com.oracle.weblogic:weblogic-maven-plugin:wlst
 -DfileName=create-datasource.py

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building maven-demo 1.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:wlst (default-cli) @ maven-demo ---
[INFO] ++=============================================================================++
[INFO] ++  weblogic-maven-plugin: wlst                                                 ++
[INFO] ++=============================================================================++

*** Creating DataSource ***

Connecting to t3://localhost:7001 with userid weblogic ...
Successfully connected to Admin Server 'AdminServer' that belongs to domain 'mydomain'.

Warning: An insecure protocol was used to connect to the
server. To ensure on-the-wire security, the SSL port or
Admin port should be used instead.

Location changed to edit tree. This is a writable tree with
DomainMBean as the root. To make changes you will need to start
an edit session via startEdit().

For more help, use help(edit)

Starting an edit session ...
Started edit session, please be sure to save and activate your
changes once you are done.
Activating all your changes, this may take a while ...
The edit lock associated with this edit session is released
once the activation is completed.
Activation completed
Location changed to serverRuntime tree. This is a read-only tree with ServerRuntimeMBean
as the root.
For more help, use help(serverRuntime)

**** DataSource Details ****

Name:            cp
Driver Name:     Oracle JDBC driver
DataSource:      oracle.jdbc.xa.client.OracleXADataSource
Properties:      {user=demo}
State:           Running
```


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

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

By default, the `wlst` goal is bound to the pre-integration-test phase. To override the default phase binding for a goal, you can explicitly bind plug-in goals to a particular life cycle phase, for example, to the post-integration-test phase, as shown below. The `pom.xml` file binds the `wlst` goal to both the pre- and post-integration-test phases (a dual phase target). As shown, you can run different scripts in different phases, overriding the default settings, and make modifications according to your needs.

Example `pom.xml` file

```
<project>
  ....
 <executions>
   <execution>
     <id>WLS_SETUP_RESOURCES</id>
     <phase>pre-integration-test</phase>
     <goals>
       <goal>wlst</goal>
     </goals>
     <configuration>
       <fileName>src/main/wlst/create-datasource.py</fileName>
     </configuration>
   </execution>

   <execution>
     <id>WLS_TEARDOWN_RESOURCES</id>
     <phase>post-integration-test</phase>
     <goals>
       <goal>wlst</goal>
     </goals>
     <configuration>
       <fileName>src/main/wlst/remove-datasource.py</fileName>
     </configuration>
   </execution>
 </executions>
 ....
 </project>
```

wlst-client

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:wlst-client`

Description

This goal is a WLST wrapper that does not require a local server install for WLST online commands. If a local server install is not present, this goal supports only WLST online commands.

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

Parameters

Table 3-21 wlst-client Parameters

Name	Type	Required	Description
`args`	`java.lang.String`	false	Deprecated. Use the scriptArgs parameter to specify the arguments as a list of scriptArg elements.
`debug`	`boolean`	false	When true, displays additional status information. Default value is: `false`
`executeScriptBeforeFile`	`boolean`	false	When true, specifies whether a script, if supplied, executes before or after the file, if supplied. Either a file or a script is required, and both are allowed. See `filename` and `script` parameters. Default value is: `true`
`failOnError`	`boolean`	false	When true, the Maven build fails if the `wlst` goal fails. The default value is `true`, and consequently any error condition will cause the build to fail. In some cases, setting `failOnError` to `false` will allow the `wlst` goal to ignore the error. Default value is: `true`
`fileName`	`java.lang.String`	false	Specifies the file path of the WLST Python script to execute. Either a `fileName` or a `script` parameter must be specified, and both are allowed.
`middlewareHome`	`java.lang.String`	false	The path to the Oracle Middleware install directory. This parameter is required for any WLST offline commands. If a WLST script uses offline commands without specifying a valid middlewareHome, this `wlst-client` goal fails.
`propertiesFile`	`java.lang.String`	false	Specifies the path to a Java properties file. The property names become defined variables in the WLST Python interpreter and are initialized to the values supplied. For example, if the properties file contains the line `"foobar: Very important stuff"`, the variable `foobar` can be used in a Python statement in the following manner: `"print('foobar has the value: ' + foobar)"`.
`script`	`java.lang.String`	false	Specifies an inline WLST Python script, for example, `"print('Hello, world!')"`. Because Python uses indentation to demarcate nested code blocks, scripts that contain multiple lines must be specified in the POM without any indentation within the pom.xml, unless required for code block demarcation.
`scriptArgs`	`java.lang.String`	false	Specifies the command-line arguments to pass to the WLST Jython interpreter as a list of string values. If the argument contains any embedded whitespace, the caller must include enclosing single quotes or escaped double quotes within the scriptArg element's value. If scriptArgs is specified, the args parameter (deprecated) is ignored.

Running Scripts With Fusion Middleware Dependencies

If you use the wlst-client goal to run WLST scripts that contain Fusion Middleware dependencies, you must first include the com.oracle.fmwshare dependency to pull in the necessary libraries needed by those scripts.

The com.oracle.fmwshare dependency must be listed before any Fusion Middleware dependencies.

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

For example, to run a WLST script for SOA, add a dependency on com.oracle.fmwshare and SOA, similar to the following:

```
<plugin>
 <groupId>com.oracle.weblogic</groupId>
 <artifactId>weblogic-maven-plugin</artifactId>
 <version>26.1.0-0-0</version>
 <executions>
  <execution>
   <id>soa-wlst-client</id>
   <goals>
    <goal>wlst-client</goal>
   </goals>
   <configuration>
    <fileName>${project.basedir}/misc/doSoaStuff.py</fileName>
    <scriptArgs>
     <scriptArg>${adminUserName}</scriptArg>
     <scriptArg>${adminPassword}</scriptArg>
     <scriptArg>${adminUrl}</scriptArg>
    </scriptArgs>
   </configuration>
  </execution>
 </executions>
 <dependencies>
  <dependency>
   <groupId>com.oracle.fmwshare</groupId>
   <artifactId>fmwshare-wlst-dependencies</artifactId>
   <version>15.1.1-0-0</version>
   <type>pom</type>
  </dependency>
  <dependency>
   <groupId>com.oracle.soa</groupId>
   <artifactId>soa-wlst-dependencies</artifactId>
   <version>15.1.1-0-0</version>
   <type>pom</type>
  </dependency>
 </dependencies>
</plugin>
```

Usage Example

The `wlst-client` goal enables the WebLogic Scripting Tool (WLST) to be used to execute scripts that configure resources or perform other operations on a WebLogic Server domain. The `wlst-client` goal does not require a local server install for WLST online commands.

The `wlst-client` Maven goal uses the WebLogic Server WLST standard environment so you can use it with all your existing WLST scripts.

You can use the `wlst-client` goal to execute an external WLST script specified with the `fileName` configuration parameter, you can specify a sequence of WLST commands within the `pom.xml` file using the `script` configuration element, or you can use both mechanisms.

For example:

```
<execution>
<id>wls-wlst-server</id>
<phase>post-integration-test</phase>
<goals>
<goal>wlst-client</goal>
</goals>
<configuration>
```


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

```
<fileName>${project.basedir}/misc/configure_resources.py</fileName>
<args>t3://some-host:7001 weblogic password AdminServer</args>
<script>
print('This is a WLST inline script\n')
print('Next, we run a WLST script to create JMS resources on the server\n')
</script>
<executeScriptBeforeFile>true</executeScriptBeforeFile>
</configuration>
</execution>
```

Example 3-20 shows typical `wlst-client` goal output.

Example 3-21 wlst-client

```
mvn com.oracle.weblogic:weblogic-maven-plugin:wlst-client
 -DfileName=create-datasource.py

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building maven-demo 1.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:wlst (default-cli) @ maven-demo ---
[INFO] ++========================================================================++
[INFO] ++  weblogic-maven-plugin: wlst                                          ++
[INFO] ++========================================================================++

*** Creating DataSource ***

Connecting to t3://some-host:7001 with userid weblogic ...
Successfully connected to Admin Server 'AdminServer' that belongs to domain 'mydomain'.

Warning: An insecure protocol was used to connect to the
server. To ensure on-the-wire security, the SSL port or
Admin port should be used instead.

Location changed to edit tree. This is a writable tree with
DomainMBean as the root. To make changes you will need to start
an edit session via startEdit().

For more help, use help(edit)

Starting an edit session ...
Started edit session, please be sure to save and activate your
changes once you are done.
Activating all your changes, this may take a while ...
The edit lock associated with this edit session is released
once the activation is completed.
Activation completed
Location changed to serverRuntime tree. This is a read-only tree with ServerRuntimeMBean
as the root.
For more help, use help(serverRuntime)

**** DataSource Details ****

Name:            cp
Driver Name:     Oracle JDBC driver
DataSource:      oracle.jdbc.xa.client.OracleXADataSource
Properties:      {user=demo}
State:           Running
```


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

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

As another example, assume that you have the following simple WLST script:

```
try:
  connect('weblogic','password','t3://10.151.69.120:7001')
  listApplications()
  print('TEST PASS')
except:
  print('TEST FAIL')
```

You can supply this WLST script with the `fileName` configuration parameter, as shown in Example 3-22.

Example 3-22 wlst-client Script Example

```
C:\Oracle\Middleware\Oracle_Home\oracle_common\plugins\maven\com\oracle\maven\or
acle-maven-sync\15.1.1>mvn com.oracle.weblogic:weblogic-maven-plugin:wlst-client
  -DfileName=test.py
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:wlst-client (default-cli) @ standalo
ne-pom ---
[INFO] [wlst-client]No middlewareHome specified.
Connecting to t3://10.151.69.120:7001 with userid weblogic ...
Successfully connected to Admin Server "AdminServer" that belongs to domain "bas
e_domain".

Warning: An insecure protocol was used to connect to the
server. To ensure on-the-wire security, the SSL port or
Admin port should be used instead.

 jaxwsejb30ws
TEST PASS
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 29.197s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2020
[INFO] Final Memory: 18M/45M
```

By default, the `wlst` goal is bound to the pre-integration-test phase. To override the default phase binding for a goal, you can explicitly bind plug-in goals to a particular life cycle phase, for example, to the post-integration-test phase, as shown below. The `pom.xml` file binds the `wlst` goal to both the pre- and post-integration-test phases (a dual phase target). As shown, you can run different scripts in different phases, overriding the default settings, and make modifications according to your needs.

Example `pom.xml` file

```
<project>
  ....
 <executions>
   <execution>
     <id>WLS_SETUP_RESOURCES</id>
     <phase>pre-integration-test</phase>
     <goals>
```


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

```
      <goal>wlst</goal>
    </goals>
    <configuration>
      <fileName>src/main/wlst/create-datasource.py</fileName>
    </configuration>
  </execution>

  <execution>
    <id>WLS_TEARDOWN_RESOURCES</id>
    <phase>post-integration-test</phase>
    <goals>
      <goal>wlst</goal>
    </goals>
    <configuration>
      <fileName>src/main/wlst/remove-datasource.py</fileName>
    </configuration>
  </execution>
</executions>
....
</project>
```

exit() is Trapped

`exit()` exits WLST from the user session and closes the scripting shell. By default, WLST calls `System.exit(0)` for the current WLST JVM when exiting WLST. Because wlst-client runs inside the same JVM as the Maven build process, the entire Maven build process would exit. To provide for this, the Maven implementation traps WLST `exit()` calls and throws an exception.

Calling exit() explicitly from a WLST script is discouraged.

For example, assume you were to modify the previous WLST script example to include `exit()`, as follows:

```
try:
  connect('weblogic','password','t3://10.151.69.120:7001')
  listApplications()
  exit()
  print('TEST PASS')
except:
  print('TEST FAIL')
```

When the Maven implementation traps `exit()`, it throws an exception:

```
Warning: An insecure protocol was used to connect to the
server. To ensure on-the-wire security, the SSL port or
Admin port should be used instead.

 jaxwsejb30ws


Exiting WebLogic Scripting Tool.

TEST FAIL
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 29.250s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2020
[INFO] Final Memory: 19M/45M
[INFO] ------------------------------------------------------------------------
```


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

ws-clientgen

Deprecated

This goal is deprecated in this release.

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:ws-clientgen`

Description

Parameters

Table 3-22 briefly describes the `ws-clientgen` parameters. These parameters are more fully described in Table 2-3 WebLogic-specific Attributes of the clientgen Ant Task in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Table 3-22 ws-clientgen Parameters

Name	Type	Required	Description
`binding` `bindings`	`java.lang.String`	false	Specifies one or more customization files that specify JAX-WS and JAXB custom binding declarations or SOAP handler files. If there is only one binding element, both `<binding>./filename</binding>` and `<bindings><binding>./filename</binding></bindings>` are allowed. See Table 3-23 for a description of `bindings` parameters.
`catalog`	`java.lang.String`	false	Specifies an external XML catalog file to resolve external entity references. For more information about creating XML catalog files, see Using XML Catalogs in *Developing JAX-WS Web Services for Oracle WebLogic Server*
`copyWsdl`	`boolean`	false	Controls where the WSDL should be copied in the ws-clientgen goal 's destination dir.
`debug`	`boolean`	false	Turns on additional debug output.
`debugLevel`	`boolean`	false	Uses Ant debug levels.
`destDir`	`java.io.File`	true	Specifies the directory into which the ws-clientgen goal generates the client source code, WSDL, and client deployment descriptor files. You must specify either the `destFile` or `destDir` attribute, but not both.
`failOnError`	`boolean`	false	Specifies whether the ws-clientgen goal continues executing in the event of an error. The default value is True.
`fork`	`boolean`	false	Specifies whether to execute javac using the JDK compiler externally. The default value is false.
`genRuntimeCatalog`	`boolean`	false	Specifies whether the ws-clientgen goal should generate the XML catalog artifacts in the client runtime environment. This value defaults to true.
`includeAntRuntime`	`boolean`	false	Specifies whether to include the Ant run-time libraries in the classpath.

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

Table 3-22 (Cont.) ws-clientgen Parameters

Name	Type	Required	Description
`includeJavaRuntime`	`boolean`	false	Specifies whether to include the default run-time libraries from the executing VM in the classpath.
`jmstransportclient`	`JMSTransportClient`	false	Invoking a WebLogic Web service using JMS transport. Table 3-25 describes the parameters of the `jmstransportclient` parameter.
`packageName`	`java.lang.String`	false	Specifies the package name into which the generated client interfaces and stub files are packaged.
`produce` `produces`	`FileSet` `List<FileSet>`	false	There is only one FileSet. There is more than one FileSet.
`verbose`	`boolean`	false	Turns on verbose output
`wsdl`	`java.lang.String`	true	Specifies a full path name or URL of the WSDL that describes a Web service (either WebLogic or non-WebLogic) for which the client component files should be generated.
`wsdlLocation`	`java.lang.String`	false	Controls the value of the wsdlLocation attribute generated on the `WebService` or `WebServiceProvider` annotation.
`xauthfile`	`java.lang.String`	false	Specifies the authorization file.
`xmlCatalog`	`java.lang.String`	false	Not used.

Table 3-23 describes the parameters of the `bindings` parameter.

Table 3-23 Binding Parameters

Name	Type	Required	Description
`file`	`java.lang.String`	false	Specifies a customization file that contains JAX-WS and JAXB custom binding declarations or SOAP handler files.

Table 3-24 describes the parameters of the `xmlCatalog` parameter.

Table 3-24 xmlCatalog Parameters

Name	Type	Required	Description
`refid`	`java.lang.String`	false	Specifies the directories (separated by semi-colons) that the `ws-jwsc` goal should search for JWS files to compile.

Table 3-25 describes the parameters of the `jmstransportclient` parameter.

Table 3-25 jmstransportclient Parameters

Name	Type	Required	Description
`destinationName`	`java.lang.String`	false	JNDI name of the destination queue or topic. Default value is `com.oracle.webservices.jms.RequestQueue`.
`destinationType`	`java.lang.String`	false	Valid values include: QUEUE or TOPIC. Default value is QUEUE.

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

Table 3-25 (Cont.) jmstransportclient Parameters

Name	Type	Required	Description
replyToName	java.lang.String	false	JNDI name of the JMS destination to which the response message is sent.
targetService	java.lang.String	false	Port component name of the Web service.
jndiInitialContextFactory	java.lang.String	false	Name of the initial context factory class used for JNDI lookup. Default value is weblogic.jndi.WLInitialContextFactory.
jndiConnectionFactoryName	java.lang.String	NA	JNDI name of the connection factory that is used to establish a JMS connection. Default value is com.oracle.webservices.jms.ConnectionFactory.
jndiUrl	java.lang.String	NA	JNDI provider URL. Default value is t3://localhost:7001.
deliveryMode	java.lang.String	NA	Delivery mode indicating whether the request message is persistent. Valid values are PERSISTENT and NON_PERSISTENT. Default value is PERSISTENT.
timeToLive	long	false	Lifetime, in milliseconds, of the request message. Default value is 180000L.
priority	int	false	JMS priority associated with the request and response message. Default value is 0.
jndiContextParameter	java.lang.String	false	JNDI properties, in a format like: someParameterName1=someValue1 , someParameterName2=someValue2.
bindingVersion	java.lang.String	false	Version of the SOAP JMS binding. Default value is 1.0.
runAsPrincipal	java.lang.String	false	Principal used to run the listening MDB.
runAsRole	java.lang.String	false	Role used to run the listening MDB.
messageType	java.lang.String	false	Message type to use with the request message. Valid values are com.oracle.webservices.api.jms.JMSMessageType.BYTES and com.oracle.webservices.api.jms.JMSMessageType.TEXT. Default value is BYTES.
enableHttpWsdlAccess	boolean	false	Boolean flag that specifies whether to publish the WSDL through HTTP. Default value is true.
mdbPerDestination	boolean	false	Boolean flag that specifies whether to create one listening message-driven bean (MDB) for each requested destination. Default value is true.
activationConfig	java.lang.String	false	Activation configuration properties passed to the JMS provider.
contextPath	java.lang.String	false	The deployed context of the web service.
serviceUri	java.lang.String	false	Web service URI portion of the URL.
portName	java.lang.String	false	The name of the port in the generated WSDL.

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

Usage Example

The `ws-clientgen` goal generates client Web service artifacts from a WSDL.

This goal benefits from the convention-over-configuration approach, allowing you to execute it using the defaults of the project.

There are two ways to run the ws-clientgen goal:

- From the command line. For example, after you define an alias:

  ```
  mvn –DvariableName1=value1 –DvariableName2=value2
  com.oracle.weblogic:weblogic-maven-plugin:ws-clientgen
  ```

- By specifying the Maven `generate-resources` life cycle phase. Then run `mvn generate-resources` in the same directory of pom.xml.

  To do this, modify the `pom.xml` file to specify the `generate-resources` life cycle phase, the `ws-clientgen` goal, and include any parameters you need to set. Consider the following example:

  ```
  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>maven_plugin.simple</groupId>
    <artifactId>maven_plugin_simple</artifactId>
    <version>1.0</version>
    <build>
      <plugins>
        <plugin>
          <groupId>com.oracle.weblogic</groupId>
          <artifactId>weblogic-maven-plugin</artifactId>
          <version>26.1.0-0-0</version>
          <executions>
            <execution>
              <id>clientgen</id>
              <phase>generate-resources</phase>
              <goals>
                <goal>ws-clientgen</goal>
              </goals>
              <configuration>
               <wsdl>${basedir}/AddNumbers.wsdl</wsdl>
                <dest${project.build.outputDirectory}</destDir>
                <packageName>maven_plugin.simple.client</packageName>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </project>
  ```

Example 3-23 shows typical `ws-clientgen` goal output.

Example 3-23 ws-clientgen

```
mvn -f C:\maven-doc\jwsc-test-2\clientgen_pom.xml generate-resources
[INFO] Scanning for projects...
```


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

```
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building maven_plugin_simple 1.0
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:ws-clientgen (clientgen) @
maven_plugin_sim
ple ---
[INFO] Executing standalone...

[INFO] Executing Maven goal 'clientgen'...
calling method public static void
weblogic.wsee.tools.clientgen.MavenClientGen.e
xecute(org.apache.maven.plugin.logging.Log,java.util.Map) throws
java.lang.Throw
able
[INFO] Consider using <depends>/<produces> so that wsimport won't do
unnecessary
 compilation
[WARNING] parsing WSDL...
[WARNING]
[WARNING]
[WARNING]
[WARNING] Generating code...
[WARNING]
[WARNING]
[WARNING] Compiling code...
[WARNING]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

wsgen

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:wsgen`

Description

Maven goal that reads a JAX-WS service endpoint implementation class and generates all of the portable artifacts for a JAX-WS Web service. Use the `wsgen` goal when you are starting from Java classes.

You can then package the service endpoint interface and implementation class, value types, and generated classes, if any, into a WAR file, and deploy the WAR to a Web container.

The `wsgen` goal provides a wrapper for the JAX-WS Maven wsgen plug-in goal.

Parameters

Table 3-26 describes the `wsgen` parameters.

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

Table 3-26 wsgen Parameters

Name	Type	Required	Description
`args`	`java.lang.String`	false	Specifies optional command-line options. Multiple elements can be specified, and each token must be placed in its own list.
`destDir`	`java.io.File`	false	Specifies the full pathname of where to place output generated classes. Use `xnocompile` to turn this off. The default is `${project.build.outputDirectory}`.
`encoding`	`java.lang.String`	false	Specifies the character encoding of the output files, such as the deployment descriptors and XML files. Examples of character encodings are SHIFT-JIS and UTF-8. The default value is platform dependent.
`extension`	`boolean`	false	`extension` is always set to `true` and you do not need to set it. Extensions are not limited to Oracle JAX-WS vendor extensions.
`executable`	`java.lang.String`	false	Name of the executable. Can be `wsgen`.
`genWsdl`	`boolean`	false	Specifies that a WSDL file should be generated in `${resourceDestDir}`. By default, the WSDL is not generated.
`inlineSchemas`	`boolean`	false	Generates inline schemas in a generated WSDL. The default is `false`. The `genWsdl` parameter must be set to `true`.
`jmstransportservice`	`boolean`	false	Use JMS transport for Web services. It can be omitted. See Table 3-34 for a description of `jmstransportservice` parameters.
`keep`	`boolean`	false	Specifies whether to keep generated files. The default is `true`.
`metadata`	`java.io.File`	false	Metadata file for the `wsgen` task, as described in External Web Service Metadata in *JAX-WS Release Documentation*. Unmatched files are ignored.
`portName`	`java.lang.String`	false	Specify the port name to use in the generated WSDL. The `genWsdl` parameter must be set to `true`.
`protocol`	`java.lang.String`	false	Use in conjunction with `genWsdl` to specify the protocol to use in the `wsdl:binding`. The `genWsdl` parameter must be set to `true`. Valid values are `soap1.1` and `Xsoap1.2`. The default is soap `soap1.1`. `Xsoap1.2` is non-standard and you can use it only in conjunction with the extension option.
`resourceDestDir`	`java.io.File`	false	Specifies the directory to contain the generated WSDL files. The default is *`${project.build.directory}`*`/generated-sources/wsdl`. The `genWsdl` parameter must be set to true.
`sei`	`java.lang.String`	false	Specifies the service endpoint implementation class name.
`servicename`	`java.lang.String`	false	Specify the service name (`wsdl:servicename`) to use in the generated WSDL. The `genWsdl` parameter must be set to true.
`sourceDestDir`	`java.io.File`	false	Specify where to place generated source files. This parameter also sets `keep` to true. The default is `${project.build.directory}/generated-sources/wsgen`.
`verbose`	`boolean`	false	Output messages about what the tool is doing. Default value is: `false`.

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

Table 3-26 (Cont.) wsgen Parameters

Name	Type	Required	Description
vmArgs	java.util.List	false	Specify optional JVM options. You can specify multiple elements, and each token must be placed in its own list.
xdonotoverwrite	boolean	false	No description provided
xnocompile	boolean	false	Turns off compilation after code generation, and lets the generated sources be compiled by Maven during the compilation phase. The default is false. This parameter also sets keep to true.

Usage Example

The wsgen goal reads a JAX-WS service endpoint implementation class and generates all of the portable artifacts for a JAX-WS Web service.

Specify the Maven process-classes life cycle phase. Then, run mvn process-classes in the same directory of the POM file.

To do this, modify the pom.xml file to specify the process-classes life cycle phase, the wsgen goal, and include any parameters you need to set. Consider the following example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven_plugin.simple</groupId>
  <artifactId>maven_plugin_simple</artifactId>
  <version>1.0</version>
  <build>
  <sourceDirectory>.</sourceDirectory>
    <plugins>
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <version>26.1.0-0-0</version>
        <executions>
          <execution>
            <id>wsgen</id>
            <phase>process-classes</phase>
            <goals>
              <goal>wsgen</goal>
            </goals>
            <configuration>
              <destDir>${project.build.directory}/wsgenOutput/</destDir>
             <sei>myexample.IPInfo</sei>
            <verbose>true</verbose>
            <genWsdl>true</genWsdl>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
```


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

```
  </build>
</project>
```

Example 3-24 shows typical wsgen goal output.

Example 3-24 wsgen

```
mvn -Dfile=pom.xml process-classes
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building maven_plugin_simple 1.0
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
maven_plug
in_simple ---
[debug] execute contextualize
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory
C:\Oracle\Middleware\Oracle_Home\orac
le_common\plugins\maven\com\oracle\maven\oracle-maven-
sync\26.1.0\src\main\resou
rces
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
maven_plugin_
simple ---
[WARNING] File encoding has not been set, using platform encoding Cp1252,
i.e. b
uild is platform dependent!
[INFO] Compiling 1 source file to
C:\Oracle\Middleware\Oracle_Home\oracle_common
\plugins\maven\com\oracle\maven\oracle-maven-sync\26.1.0\target\classes
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:wsgen (wsgen) @
maven_plugin_simple
---
[INFO] Processing: myexample.IPInfo
[WARNING] Using platform encoding (Cp1252), build is platform dependent!
[INFO] jaxws:wsgen args: [-keep, -s,
'C:\Oracle\Middleware\Oracle_Home\oracle_co
mmon\plugins\maven\com\oracle\maven\oracle-maven-sync\15.1.1\target\generated-
so
urces\wsgen', -d,
'C:\Oracle\Middleware\Oracle_Home\oracle_common\plugins\maven\
com\oracle\maven\oracle-maven-sync\26.1.0\target\wsgenOutput', -verbose, -
extens
ion, -wsdl, -r,
'C:\Oracle\Middleware\Oracle_Home\oracle_common\plugins\maven\co
m\oracle\maven\oracle-maven-sync\26.1.0\target\generated-sources\wsdl',
myexampl
```


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

```
e.IPInfo]
myexample\jaxws\GetIpAddress.java
myexample\jaxws\GetIpAddressResponse.java
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 21.309s
[INFO] Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 8M/32M
[INFO]
------------------------------------------------------------------------
```

In this example, the `wsgen` goal creates the following files:

```
target
  classes
   META-INF
      wsdl
        IPInfoService.wsdl
        IPInfoService_schema1.xsd
      myexample
        IPInfo.class
  generated-sources
      wsdl
        IPInfoService.wsdl
        IPInfoService_schema1.xsd
      wsgen
        myexample
          jaxws
            GetIpAddress.java
            GetIpAddressResponse.java
  wsgenoutput
        myexample
          jaxws
            GetIpAddress.class
            GetIpAddressResponse.class
```

wsimport

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:wsimport`

Description

Maven goal that parses a WSDL and binding files and generates the Java code needed to access it. Use the `wsimport` goal when you are starting from a WSDL.

The `wsimport` goal provides a wrapper for the JAX-WS Maven wsimport goal.

Parameters

Table 3-27 describes the `wsimport` parameters.

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

Table 3-27 wsimport Parameters

Name	Type	Required	Description
`args`	`java.lang.String`	false	Specifies optional command-line options. Multiple elements can be specified, and each token must be placed in its own list.
`bindingDirectory`	`java.io.File`	false	Directory containing binding files.
`bindingFiles`	`java.util.List`	false	List of files to use for bindings. If not specified, all `.xml` files in the `bindingDirectory` are used.
`catalog`	`java.io.File`	false	Catalog file to resolve external entity references support TR9401, XCatalog, and OASIS XML Catalog format.
`destDir`	`java.io.File`	false	Specifies the full pathname of where to place output generated classes. Use `xnocompile` to turn this off. The default is ${project.build.outputDirectory}).
`encoding`	`java.lang.String`	false	Specifies the character encoding of the output files, such as the deployment descriptors and XML files. Examples of character encodings are SHIFT-JIS and UTF-8. The default is platform dependent.
`executable`	`java.lang.String`	false	Name of the executable. Can be `wsimport`.
`extension`	`boolean`	false	`extension` is always set to `true` and you do not need to set it. Extensions are not limited to Oracle JAX-WS vendor extensions.
`genJWS`	`boolean`	false	Generate stubbed JWS implementation file. The default is `false`.
`httpproxy`	`java.lang.String`	false	Set HTTP/HTTPS proxy. Format is `[user[:password]@]proxyHost[:proxyPort]`.
`implDestDir`	`java.io.File`	false	Specify where to generate JWS implementation file.
`implPortName`	`java.lang.String`	false	Local portion of port name for generated JWS implementation. Implies `genJWS=true`. Note: It is a QName string, formatted as: "{" + Namespace URI + "}" + local part.
`implServiceName`	`java.lang.String`	false	Local portion of service name for generated JWS implementation. Implies `genJWS=true`. Note: It is a QName string, formatted as: "{" + Namespace URI + "}" + local part.
`jmstransportclient`	`JMSTransportClient`	false	Invoking a WebLogic Web service using JMS transport. Table 3-25 describes the parameters of the `jmstransportclient` parameter.
`jmsUri`	`jmsUri`	false	Override jmsUri defined in a WSDL file. Requires `extension=true`.
`keep`	`boolean`	false	Specifies whether to keep generated files. The default is `true`.
`packageName`	`java.lang.String`	false	The package in which the source files will be generated.
`quiet`	`boolean`	false	Suppress wsimport output. The default is `false`.
`sourceDestDir`	`java.io.File`	false	Specify where to place generated source files. This parameter also sets `keep` to true. The default is ${project.build.directory}/generated-sources/wsimport.
`staleFile`	`java.io.File`	false	The folder containing flag files used to determine if the output is stale. If you do not specify a folder, the default is ${project.build.directory}/jaxws/stale.

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

Table 3-27 (Cont.) wsimport Parameters

Name	Type	Required	Description
target	java.lang.String	false	Generate code as per the given JAXWS specification version. Setting "2.0" will cause JAX-WS to generate artifacts that run with JAX-WS 2.0 runtime.
verbose	boolean	false	Output messages about what the tool is doing. Default value is: false.
vmArgs	java.lang.String	false	Specify optional JVM options. You can specify multiple elements, and each token must be placed in its own list.
wsdlDirectory	java.io.File	false	Directory containing WSDL files.
wsdlFiles	java.util.List	false	List of files to use for WSDLs. If not specified, all .wsdl files in the wsdlDirectory will be used.

wsdlLocation	java.lang.String	false	@WebService.wsdlLocation and @WebServiceClient.wsdlLocation value.
			Can end with asterisk, in which case relative path of the WSDL will be appended to the given wsdlLocation.
			Example:
			`...`
			`<configuration>`
			`<wsdlDirectory>src/mywsdls</wsdlDirectory>`
			`<wsdlFiles>`
			`<wsdlFile>a.wsdl</wsdlFile>`
			`<wsdlFile>b/b.wsdl</wsdlFile>`
			`<wsdlFile>${basedir}/src/mywsdls/c.wsdl</wsdlFile>`
			`</wsdlFiles>`
			`<wsdlLocation>http://example.com/mywebservices/*</wsdlLocation>`
			`</configuration>`
			`...`
			wsdlLocation for a.wsdl will be http://example.com/mywebservices/a.wsdl
			wsdlLocation for b/b.wsdl will be http://example.com/mywebservices/b/b.wsdl
			wsdlLocation for ${basedir}/src/mywsdls/c.wsdl will be file://absolute/path/to/c.wsdl
			Note: External binding files cannot be used if asterisk notation is in place.

wsdlUrls	java.util.List	false	List of external WSDL URLs to be compiled.
xadditionalHeaders	boolean	false	Maps headers not bound to the request or response messages to Java method parameters.
xauthFile	java.io.File	false	Specify the location of authorization file.
xdebug	boolean	false	Turn on debug message. The default is false.
xdisableAuthenticator	boolean	false	Disable Authenticator used by JAX-WS RI, xauthfile will be ignored if set.

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

Table 3-27 (Cont.) wsimport Parameters

Name	Type	Required	Description
xdisableSSLHostnameVerification	boolean	false	Disable the SSL Hostname verification while fetching WSDL(s).
xjcArgs	java.util.List	false	Specify optional XJC-specific parameters that should simply be passed to xjc using -B option of WsImport command. Multiple elements can be specified, and each token must be placed in its own list.
xnoAddressingDataBinding	boolean	false	Binding W3C EndpointReferenceType to Java. By default WsImport follows spec and does not bind EndpointReferenceType to Java and uses the spec provided W3CEndpointReference.
xnocompile	boolean	false	Turns off compilation after code generation, and lets the generated sources be compiled by Maven during the compilation phase. The default is true. This parameter also sets keep to true.
xuseBaseResourceAndURLToLoadWSDL	boolean	false	No description provided by JAX-WS Maven wsimport.

Usage Example

The wsimport goal parses a WSDL and binding files and generates Java code needed to access the Web service.

You can use the wsimport goal in two ways:

- To generate the client-side artifacts. Then, implement the client to invoke the Web service.
- To create your own implementation of the Web service. Use wsimport goal with the genJWS parameter to generate portable artifacts and a stubbed implementation file. You then implement the service endpoint.

Specify the Maven generate-sources life cycle phase. Then, run mvn generate-sources in the same directory of the POM file.

Assume that you want to import the WSDL shown in Example 3-25.

Example 3-25 WSDL to Import

```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at
 http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.9-b14041
 svn-revision#14041. --><!-- Generated by JAX-WS RI at
 http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.9-b14041
 svn-revision#14041. --><definitions
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
uti
lity-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1_
2="http://schemas.xmlsoap.org/ws/2004/09/policy"
 xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://ws.web.wls.my.org/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://ws.web.wls.my.org/" name="SampleWs">
```


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

```
    <types>
        <xsd:schema>
            <xsd:import namespace="http://ws.web.wls.my.org/"
schemaLocation="x.xsd"/>
        </xsd:schema>
    </types>
    <message name="hello">
        <part name="parameters" element="tns:hello"/>
    </message>
    <message name="helloResponse">
        <part name="parameters" element="tns:helloResponse"/>
    </message>
    <portType name="SampleWs">
        <operation name="hello">
            <input wsam:Action="http://ws.web.wls.my.org/SampleWs/
helloRequest" message="tns:hello"/>
            <output wsam:Action="http://ws.web.wls.my.org/SampleWs/
helloResponse" message="tns:helloResponse"/>
        </operation>
    </portType>
    <binding xmlns:soapjms="http://www.w3.org/2010/soapjms/"
name="SampleWsPortBinding" type="tns:SampleWs">

<soapjms:jndiInitialContextFactory>weblogic.jndi.WLInitialContextFactory</
soapjms:jndiInitialContextFactory>

<soapjms:jndiConnectionFactoryName>com.oracle.webservices.api.jms.ConnectionFa
ctory</soapjms:jndiConnectionFactoryName>
        <soapjms:jndiUrl>t3://localhost:7001</soapjms:jndiUrl>
        <soapjms:bindingVersion>SOAP_JMS_1_0</soapjms:bindingVersion>
        <soapjms:destinationName>com.oracle.webservices.api.jms.RequestQueue</
soapjms:destinationName>
        <soapjms:targetService>SampleWs</soapjms:targetService>
        <soapjms:timeToLive>180000</soapjms:timeToLive>
        <soapjms:deliveryMode>PERSISTENT</soapjms:deliveryMode>
        <soapjms:priority>0</soapjms:priority>
        <soapjms:messageType>BYTES</soapjms:messageType>
        <soapjms:destinationType>QUEUE</soapjms:destinationType>
        <soap:binding transport="http://www.w3.org/2010/soapjms/"
style="document"/>
        <operation name="hello">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="SampleWs">
        <port name="SampleWsPort" binding="tns:SampleWsPortBinding">
            <soap:address
location="jms:jndi:com.oracle.webservices.api.jms.RequestQueue?
targetService=Sampl
eWs&amp;jndiURL=t3://
```


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

```
localhost:7001&amp;messageType=BYTES&amp;deliveryMode=PERSISTENT"/>
        </port>
    </service>
</definitions>
```

To import this WSDL, modify the `pom.xml` file to specify the `generate-sources` life cycle phase, the `wsimport` goal, the WSDL location, and include any parameters you need to set. This example uses a local WSDL file for demonstration purposes.

Consider the following example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven_plugin.simple</groupId>
  <artifactId>maven_plugin_simple</artifactId>
  <version>1.0</version>
  <build>
   <plugins>
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <version>26.1.0-0-0</version>
        <executions>
          <execution>
          <id>wsimport-jmssample</id>
           <goals>
           <goal>wsimport</goal>
           </goals>
            <phase>generate-sources</phase>
            <configuration>
               <wsdlFiles>
                 <wsdlFile>${basedir}/import-example/SampleWs.wsdl</wsdlFile>
               </wsdlFiles>
               <genJWS>true</genJWS>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Example 3-26 shows typical `wsimport` goal output.

Example 3-26 wsimport

```
mvn -Dfile=pom.xml generate-sources
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building maven_plugin_simple 1.0
[INFO]
------------------------------------------------------------------------
```


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

```
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:wsimport (wsimport-jmssample) @
mave
n_plugin_simple ---
[INFO] Processing: file:/C:/Oracle/Middleware..../import-example/SampleWs.wsdl
[WARNING] Using platform encoding (Cp1252), build is platform dependent!
[INFO] jaxws:wsimport args: [-keep, -s,
'C:\Oracle\Middleware\...\import-example\target\generated-sources\wsimport', -
d,
 'C:\Oracle\Middleware...\import-example\target\classes', -extension,
 -Xnocompile, -jms, -jmsuri, jms:jndi:null?targetServi
ce=null, -httpproxy:some-proxy-name, -generateJWS, -implDestDir,
'C:\Oracle\Middleware...\import-example',
"file:/C:/Oracle/Middleware...import-example/SampleWs.wsdl"]
parsing WSDL...



Generating code...

[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 20.888s
[INFO] Finished at: Finished at: Wed Aug 19 11:33:51 EDT 2015
[INFO] Final Memory: 7M/23M
[INFO]
------------------------------------------------------------------------
```

In this example, the `wsimport` goal creates the following files:

```
org
  my
   wls
    web
     ws
      SampleWs_SampleWsPortImpl.java
target
  classes
  generated-sources
   wsimport
    org
     my
      wls
       web
        ws
         Hello.java
         HelloResponse.java
         ObjectFactory.java
         package-info.java
         SampleWs.java
         SampleWs_Service.java
  jaxws
```


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

```
stale
 .2b48c6ef28bc8a45aa2da4246c0c4ac90cf82c57
```

ws-wsdlc

Deprecated

This goal is deprecated in this release.

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:ws-wsdlc`

Description

Maven goal to generate a set of artifacts and a partial Java implementation of the Web service from a WSDL.

The `ws-wsdlc` goal provides a Maven wrapper for the wsdlc Ant task, which is described in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Parameters

Table 3-28 briefly describes the `ws-wsdlc` parameters. These parameters are more fully described in Table 2-3 WebLogic-specific Attributes of the clientgen Ant Task in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Table 3-28 ws-wsdlc Parameters

Name	Type	Required	Description
`bindings`	`java.lang.String`	false	Customization files that specify JAX-WS and JAXB custom binding declarations or SOAP handler files.
`catalog`	`java.lang.String`	false	Specifies an external XML catalog file. For more information about creating XML catalog files, see Using XML Catalogs in *Developing JAX-WS Web Services for Oracle WebLogic Server*
`debug`	`boolean`	false	Specifies the flag to set when debugging the process. Default value is false.
`debugLevel`	`java.lang.String`	false	Uses Ant debug levels.
`destImplDir`	`java.lang.String`	false	Specifies the directory into which the stubbed-out JWS implementation file is generated.
`destJavadocDir`	`java.lang.String`	false	Specifies the directory into which the Javadoc that describes the JWS interface is generated.
`destJwsDir`	`java.lang.String`	true	Specifies the directory into which the JAR file that contains the JWS interface and data binding artifacts should be generated.
`explode`	`boolean`	false	Specifies the flag to set if you want exploded output. Defaults to true.
`failOnError`	`boolean`	false	Specifies whether the ws-clientgen goal continues executing in the event of an error. The default value is true
`fork`	`boolean`	false	Specifies whether to execute javac using the JDK compiler externally. The default value is false.

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

Table 3-28 (Cont.) ws-wsdlc Parameters

Name	Type	Required	Description
includeAntRuntime	boolean	false	Specifies whether to include the Ant run-time libraries in the classpath. The default value is true.
includeJavaRuntime	boolean	false	Specifies whether to include the default run-time libraries from the executing VM in the classpath. The default value is false.
optimize	boolean	false	Specifies the flag to set if you want optimization. Defaults to true.
packageName	java.lang.String	false	Specifies the package into which the generated JWS interface and implementation files should be generated.
srcPortName	java.lang.String	false	Specifies the name of the WSDL port from which the JWS interface file should be generated. Set the value of this parameter to the value of the name parameter of the port parameter that corresponds to the Web service port for which you want to generate a JWS interface file. The port parameter is a child of the service parameter in the WSDL file. If you do not specify this attribute, ws-wsdlc generates a JWS interface file from the service specified by srcServiceName.
srcServiceName	java.lang.String	false	Specifies the name of the Web service from which the JWS interface file should be generated.
srcWsdl	java.lang.String	true	Specifies the name of the WSDL from which to generate the JAR file that contains the JWS interface and data binding artifacts.
verbose	boolean	false	Specifies the flag to set if you want verbose output. Default value is false.

Usage Example

The ws-wsdlc goal generates a set of artifacts and a partial Java implementation of the Web service from a WSDL.

This goal benefits from the convention-over-configuration approach, allowing you to execute it using the defaults of the project.

There are two ways to run the ws-wsdlc goal:

- From the command line. For example, after you define an alias:

  ```
  mvn –DvariableName1=value1 –DvariableName2=value2
  com.oracle.weblogic:weblogic-maven-plugin:ws-wsdlc
  ```

- By specifying the Maven generate-resources life cycle phase.

  To do this, modify the pom.xml file to specify the generate-resources life cycle phase, the ws-wsdlc goal, and include any parameters you need to set. Then run mvn generate-resources in the same directory of pom.xml.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven_plugin.simple</groupId>
  <artifactId>maven_plugin_simple</artifactId>
  <version>1.0</version>
  <build>
    <plugins>
```


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

```
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <version>26.1.0-0-0</version>
        <executions>
          <execution>
            <id>wsdlc</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>ws-wsdlc</goal>
            </goals>
            <configuration>
              <srcWsdl>${basedir}/AddNumbers.wsdl</srcWsdl>
              <destJwsDir>${project.build.directory}/jwsImpl</destJwsDir>
              <destImplDir>${project.build.directory}/output</destImplDir>
              <packageName>maven_plugin.simple</packageName>
              <verbose>true</verbose>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Example 3-27 shows typical `ws-wsdlc` goal output.

Example 3-27 ws-wsdlc

```
mvn -f wsdlc_pom.xml generate-resources
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building maven_plugin_simple 1.0
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:ws-wsdlc (wsdlc) @
maven_plugin_simple ---
[INFO] Executing standalone...

[INFO] Executing Maven goal 'wsdlc'...
calling method public static void
weblogic.wsee.tools.wsdlc.MavenWsdlc.execute(o
rg.apache.maven.plugin.logging.Log,java.util.Map) throws java.lang.Throwable
Catalog dir = C:\Users\maven\AppData\Local\Temp\_ckr59b
Download file [AddNumbers.wsdl] to C:\Users\maven\AppData\Local\Temp\_ckr59b
srcWsdl is redefined as [ C:\Users\maven\AppData\Local\Temp\_ckr59b\AddNumber
s.wsdl ]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```


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

ws-jwsc

Deprecated

This goal is deprecated in this release.

Full Name

`com.oracle.weblogic:weblogic-maven-plugin:ws-jwsc`

Description

Maven goal to build a JAX-WS web service.

The `ws-jwsc` goal provides a Maven wrapper for the jwsc Ant task, which is described in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Nested Configuration in module Elements

The `ws-jwsc` goal supports nested configuration elements, as shown in bold in Example 3-28. See Introduction to the POM for information on Maven projects with multiple modules.

Example 3-28 Nested Configuration Elements

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test.ws</groupId>
  <artifactId>test-ws-jwsc1</artifactId>
  <version>1.0</version>
  <build>
    <plugins>
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <version>26.1.0-0-0</version>
        <executions>
          <execution>
            <id>first-jwsc</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>ws-jwsc</goal>
            </goals>
            <configuration>
              <srcDir>${basedir}/src/main/java</srcDir>
            <destDir>${project.build.directory}/jwscOutput
                  /${project.build.finalName}</destDir>
              <listfiles>true</listfiles>
              <debug>true</debug>

                <module>
                  <name>pocreate</name>
                  <contextPath>mypub</contextPath>
                  <compiledWsdl>D:\maven-test\order_wsdl.jar</compiledWsdl >

                    <jws>
```


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

```
                        <file>examples/wsee/jwsc/POCreateImpl.java</file>
                         <transportType>
                           <type>WLHttpTransport</type>
                           <serviceUri>POCreate</serviceUri>
                           <portName>POCreatePort</portName>
                         </transportType>
                       </jws>
                       <jws>
                         …
                       </jws>
                      <descriptors>
                        <descriptor>"resources/web.xml"<descriptor/>
                        <descriptor>"resources/weblogic.xml"<descriptor />
                      </descriptors>
                    </module>
                    <module>
                    …
                    </module>
                  </modules>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </project>
```

These nested configuration elements for `ws-jwsc` have the following conditions:

- You must use at least one of the following elements: `jws`, `jwses`, `module`, or `modules`.
- Collection elements such as `jwses` and `modules` elements can be omitted.
- If there is only one child element within the collection element, the collection element can also be removed.

  For example, if there is only one `jws` element, use `jws`. If there are multiple `jws` elements, add all of the `jws` elements under a `jwses` element.
- As with the JWSC ant task, if `module` has only one `jws` child element, then other sub elements of `module` can be nested into `jwsc` and `jwsc/transportType`.

Example 3-29 shows an example without a module `element` in which the `jws` parameter is a child of `ws-jwsc`.

Example 3-29 jws Element as Child of ws-jwsc Goal

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test.ws</groupId>
  <artifactId>test-ws-jwsc</artifactId>
  <version>1.0</version>
  <build>
    <plugins>
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
```


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

```
            <version>26.1.0-0-0</version>
            <executions>
              <execution>
                <id>first-jwsc</id>
                <phase>compile</phase>
                <goals>
                  <goal>ws-jwsc</goal>
                </goals>
                <configuration>
                  <srcDir>${basedir}/src/main/java</srcDir>
                  <destDir>${project.build.directory}/jwscOutput/
                      ${project.build.finalName}</destDir>
                  <jws>            <!-- no parent <module> -->
                    <file>examples/wsee/jwsc/POCreateImpl.java</file>
                     <compiledWsdl>${project.build.directory}/
purchaseorder_wsdl.jar>
                    <transportType>
                      <type>WLHttpTransport</type>
                    </transportType>
                  </jws>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </project>
```

ws-jwsc Parameters

Table 3-29 briefly describes the `ws-jwsc` parameters. These parameters are more fully described in Table 2-3 WebLogic-specific Attributes of the clientgen Ant Task in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Table 3-29 ws-jwsc Parameters

Name	Type	Required	Description
`applicationXml`	`java.lang.String`	false	Specifies the full name and path of the application.xml deployment descriptor of the Enterprise Application. If you specify an existing file, the ws-jwsc goal updates it to include the Web services information. However, jwsc does not automatically copy the updated application.xml file to the destDir; you must manually copy this file to the destDIR. If the file does not exist, jwsc creates it. The ws-jwsc goal also creates or updates the corresponding weblogic-application.xml file in the same directory. If you do not specify this attribute, jwsc creates or updates the file destDir/META-INF/application.xml, where destDir is the jwsc attribute.
`debug`	`boolean`	false	Turns on additional debug output.
`destDir`	`java.lang.String`	true	Specifies the full pathname of the directory that will contain the compiled JWS files, XML Schemas, WSDL, and generated deployment descriptor files, all packaged into a JAR or WAR file.
`destEncoding`	`java.lang.String`	false	Specifies the character encoding of the output files, such as the deployment descriptors and XML files. Examples of character encodings are SHIFT-JIS and UTF-8. The default value of this attribute is UTF-8.

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

Table 3-29 (Cont.) ws-jwsc Parameters

Name	Type	Required	Description
`jws`	`Jws`	false	There is only one <jws> element. See Table 3-30 for a description of `jws` parameters.
`jwses`	`Jws`	false	It contains more than one< jws> element.
`keepGenerated`	`boolean`	false	Specifies whether the Java source files and artifacts generated by this goal should be regenerated if they already exist. If you specify false, new Java source files and artifacts are always generated and any existing artifacts are overwritten. If you specify true, the goal regenerates only those artifacts that have changed, based on the timestamp of any existing artifacts
`listfiles`	`boolean`	false	Specifies whether to list all of the files.
`module`	`Module`	false	It contains one <module> element. See Table 3-31 for a description of `module` parameters.
`modules`	`Module`	false	It contains more than one <module> element.
`optimize`	`boolean`	false	Specifies the flag to set when optimization is required. Defaults to true.
`sourcepath`	`java.lang.String`	true	The full pathname of top-level directory that contains the Java files referenced by the JWS file, such as JavaBeans used as parameters or user-defined exceptions.
`srcDir`	`java.lang.String`	true	Specifies the full pathname of the top-level directory that contains the JWS file you want to compile.
`srcEncoding`	`java.lang.String`	false	Specifies the character encoding of the input files, such as the JWS file or configuration XML files. Examples of character encodings are SHIFT-JIS and UTF-8. The default value of this attribute is the character encoding set for the JVM.
`verbose`	`boolean`	false	Specifies verbose output

jws Parameter

As described in jws, the `jws` parameter specifies the name of a JWS file that implements your Web service and for which the `ws-jwsc` goal should generate Java code and supporting artifacts, and then package them into a deployable WAR file inside of an Enterprise Application.

You can specify the `jws` parameter in two ways:

- An immediate child element of the `ws-jwsc` goal. In this case, `ws-jwsc` generates a separate WAR file for each JWS file. You typically use this method if you are specifying just one JWS file to the `ws-jwsc` goal.
- A child element of the `module` parameter, which in turn is a child of the `ws-jwsc` goal. In this case, `ws-jwsc` generates a single WAR file that includes all the generated code and artifacts for all the JWS files grouped within the module parameter.

  This method is useful if you want all JWS files to share supporting files, such as common Java data types.

Table 3-30 describes the child parameters of the `jws` parameter. The description specifies whether the parameter applies in the case that `jws` is a child of the `ws-jwsc` goal, is a child of `module`, or both.

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

Table 3-30 jws Parameters

Name	Type	Required	Description	Child of ws-jwsc, module, or both
`compiledWsdl`	`java.lang.String`	false	Specifies the full pathname of the JAR file generated by the `ws-wsdlc` goal based on an existing WSDL file. Only required for the "starting from WSDL" use case.	both
`contextPath`	`java.lang.String`	false	Specifies the deployed context of the web service.	ws-jwsc
`explode`	`boolean`	false	Specifies the flag to set when you want exploded output. Defaults to true.	ws-jwsc
`file`	`java.lang.String`	true	The name of the JWS file that you want to compile. The ws-jwsc goal looks for the file in the srcdir directory.	both
`generateWsdl`	`boolean`	true	Specifies whether the generated WAR file includes the WSDL file in the WEB-INF directory. Default value is false.	both
`jmstransportservice`	`boolean`	false	Use JMS transport for Web services. It can be omitted. See Table 3-34 for a description of `jmstransportservice` parameters.	ws-jwsc
`name`	`java.lang.String`	false	Specifies the name of the generated WAR file (or exploded directory, if the explode attribute is set to true) that contains the deployable Web service.	ws-jwsc
`transportType`	`transportType`	false	Used when it contains only one transport `type` element. It can be omitted. See Table 3-33 for a description of `transportType` parameters.	both
`transportTypes`	`transportType`	false	Used when it contains more than one transport `type` element. It can be omitted. See Table 3-33 for a description of `transportType` parameters.	both
`wsdlOnly`	`boolean`	false	Specifies that only a WSDL file should be generated for this JWS file. The default value is false.	ws-jwsc

module Parameters

As described in module, the `module` parameter groups one or more `jws` parameters together so that their generated code and artifacts are packaged in a single Web application (WAR) file. The `module` parameter is a child of the `ws-jwsc` goal.

Table 3-31 describes the parameters of the module parameter.

Table 3-31 module Parameters

Name	Type	Required	Description
`clientgen`	`java.lang.String`	false	There is only one <clientgen> element. It can be omitted.
`clientgens`	`java.lang.String`	false	There is more than one <clientgen> element. It can be omitted.

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

Table 3-31 (Cont.) module Parameters

Name	Type	Required	Description
contextPath	java.lang.String	false	Specifies the deployed context of the Web service.
descriptor	java.lang.String	false	Specifies the web.xml descriptor to use if a new one should not be generated. The path should be fully qualified. The files should be separated by ", ".
ejbWsInWar	boolean	false	Specifies whether to package EJB-based Web services in a WAR file instead of a JAR file.
explode	boolean	false	Specifies the flag to set when you want exploded output. Defaults to true.
FileSet	FileSet	false	Used when it contains one FileSet element. It can be omitted.
FileSets	FileSet	false	Used when it contains more than one FileSet element. It can be omitted.
generateWsdl	boolean	true	Specifies whether the generated WAR file includes the WSDL file in the WEB-INF directory. Default value is false.
jws	Jws	false	Used when it contains one jws element. It can be omitted.
jwses	Jws	false	Used when it contains more than one jws element. It can be omitted.
name	java.lang.String	false	Specifies the name of the WAR to use when evaluating the ear file.
wsdlOnly	boolean	false	Specifies that only a WSDL file should be generated for this JWS file. The default value is false.
zipfileset	java.lang.String	false	There is only one <zipfileset> element.

FileSet Parameters

As described in jwsfileset, the `FileSet` parameter specifies one or more directories in which the `ws-jwsc` goal searches for JWS files to compile. The list of JWS files that `ws-jwsc` finds is then treated as if each file had been individually specified with the `jws` parameter of `module`.

The `FileSet` parameter is a child of the `ws-jwsc` goal.

Table 3-32 describes the parameters of the FileSet parameter.

Table 3-32 FileSet Parameters

Name	Type	Required	Description
srcDir	java.lang.String	true	Specifies the directories (separated by semi-colons) that the ws-jwsc goal should search for JWS files to compile.
prefix	java.lang.String	false	Prefix to use.
sourceIncludes	java.lang.String	false	Specifies the explicit includes-list for the file set.
sourceExcludes	java.lang.String	false	Specifies the explicit excludes-list for the file set.

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

TransportType Parameters

As described in WLHttpTransport, WLHttpsTransport, and WLJMSTransport, you use transport parameters to specify the transport type, context path, and service URI sections of the URL used to invoke the Web service, as well as the name of the port in the generated WSDL.

The `ws-jwsc` goal combines these transport parameters into one, `TransportType`.

Table 3-32 describes the parameters of the `transportType` parameter.

Table 3-33 transportType Parameters

Name	Type	Required	Description
`transportTypeName`	`java.lang.String`	true	Specifies the value is WLHttpTransport, WLHttpsTransport, or WLJMSTransport. Default value is WLHttpTransport.
`serviceUri`	`java.lang.String`	false	Specifies the Web service URI portion of the URL.
`contextPath`	`java.lang.String`	false	Specifies the deployed context of the Web service.
`portName`	`java.lang.String`	false	Specifies the name of the port in the generated WSDL.

Table 3-34 describes the parameters of the `jmstransportservice` parameter.

Table 3-34 jmstransportservice Parameters

Name	Type	Required	Description
`destinationName`	`java.lang.String`	false	JNDI name of the destination queue or topic. Default value is `com.oracle.webservices.jms.RequestQueue`.
`destinationType`	`java.lang.String`	false	Valid values include: QUEUE or TOPIC. Default value is QUEUE.
`replyToName`	`java.lang.String`	false	JNDI name of the JMS destination to which the response message is sent.
`targetService`	`java.lang.String`	false	Port component name of the Web service.
`jndiInitialContextFactory`	`java.lang.String`	false	Name of the initial context factory class used for JNDI lookup. Default value is `weblogic.jndi.WLInitialContextFactory`.
`jndiConnectionFactoryName`	`java.lang.String`		JNDI name of the connection factory that is used to establish a JMS connection. Default value is `com.oracle.webservices.jms.ConnectionFactory`.
`jndiUrl`	`java.lang.String`		JNDI provider URL. Default value is `t3://localhost:7001`.
`deliveryMode`	`java.lang.String`		Delivery mode indicating whether the request message is persistent. Valid values are PERSISTENT and NON_PERSISTENT. Default value is PERSISTENT.
`timeToLive`	`long`	false	Lifetime, in milliseconds, of the request message. Default value is 180000L.
`priority`	`int`	false	JMS priority associated with the request and response message. Default value is 0.

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

Table 3-34 (Cont.) jmstransportservice Parameters

Name	Type	Required	Description
jndiContextParameter	java.lang.String	false	JNDI properties, in a format like: someParameterName1=someValue1 , someParameterName2=someValue2.
bindingVersion	java.lang.String	false	Version of the SOAP JMS binding. Default value is 1.0.
runAsPrincipal	java.lang.String	false	Principal used to run the listening MDB.
runAsRole	java.lang.String	false	Role used to run the listening MDB.
messageType	java.lang.String	false	Message type to use with the request message. Valid values are com.oracle.webservices.api.jms.JMSMessageType.BYTES and com.oracle.webservices.api.jms.JMSMessageType.TEXT. Default value is BYTES.
enableHttpWsdlAccess	boolean	false	Boolean flag that specifies whether to publish the WSDL through HTTP. Default value is true.
mdbPerDestination	boolean	false	Boolean flag that specifies whether to create one listening message-driven bean (MDB) for each requested destination. Default value is true.
activationConfig	java.lang.String	false	Activation configuration properties passed to the JMS provider.
contextPath	java.lang.String	false	The deployed context of the web service.
serviceUri	java.lang.String	false	Web service URI portion of the URL.
portName	java.lang.String	false	The name of the port in the generated WSDL.

Usage Example

The ws-jwsc goal builds a JAX-WS web service.

This goal benefits from the convention-over-configuration approach, allowing you to execute it using the defaults of the project.

To run the ws-jwsc goal, specify the Maven generate-resources phase.

To do this, modify the pom.xml file to specify the generate-resources phase, the ws-jwsc goal, and include any pa parameters you need to set. Then run mvn generate-resources in the same directory of pom.xml.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven_plugin.simple</groupId>
  <artifactId>maven_plugin_simple</artifactId>
  <version>1.0</version>
  <build>
    <plugins>
```


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

```
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <version>26.1.0-0-0</version>
        <executions>
          <execution>
            <id>jwsc</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>ws-jwsc</goal>
            </goals>
            <configuration>
              <destDir>${project.build.directory}/jwscOutput/
              <listfiles>true</listfiles>
              <debug>true</debug>
              <jws>            <!-- no parent <module> -->
                <file>examples/wsee/jwsc/POCreateImpl.java</file>
                  <compiledWsdl>${project.build.directory}/
purchaseorder_wsdl.jar>
                <transportType>
                  <type>WLHttpTransport</type>
                </transportType>
</jws>
              <verbose>true</verbose>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Example 3-30 shows typical `ws-jwsc` goal output.

Example 3-30 ws-jwsc

```
mvn -f jwsc_pom.xml generate-resources
INFO] Scanning for projects...
[INFO]

[INFO]
------------------------------------------------------------------------
[INFO] Building maven_plugin_simple 1.0
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- weblogic-maven-plugin:26.1.0-0-0:ws-jwsc (jwsc) @
maven_plugin_simple ---
[INFO] Executing standalone...

INFO] Executing Maven goal 'jwsc'...
calling method public static void
weblogic.wsee.tools.jws.MavenJwsc.execute(org.apache.maven.plugin.logging.Log,
java.util.Map) throws java.lang.Throwable
[EarFile] Application File : C:\maven-doc\jwsc-test-2\output\META-
INF\application.xml
```


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

```
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```


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

4

Creating a Split Development Directory Environment

To create a WebLogic Server split development directory for a Jakarta EE enterprise application, organize the application modules and shared classes, then generate and customize a build.xml file. If your project contains multiple EARs, you can coordinate their builds as described later in this chapter.

Overview of the Split Development Directory Environment

The WebLogic split development directory environment consists of a directory layout and associated Ant tasks that help you repeatedly build, change, and deploy Jakarta EE applications.

Compared to other development frameworks, the WebLogic split development directory provides these benefits:

- **Fast development and deployment.** By minimizing unnecessary file copying, the split development directory Ant tasks help you recompile and redeploy applications quickly *without* first generating a deployable archive file or exploded archive directory.
- **Simplified build scripts.** The Oracle-provided Ant tasks automatically determine which Jakarta EE modules and classes you are creating, and build components in the correct order to support common classpath dependencies. In many cases, your project build script can simply identify the source and build directories and allow Ant tasks to perform their default behaviors.
- **Easy integration with source control systems.** The split development directory provides a clean separation between source files and generated files. This helps you maintain only editable files in your source control system. You can also clean the build by deleting the entire build directory; build files are easily replaced by rebuilding the project.

Source and Build Directories

The source and build directories form the basis of the split development directory environment. The source directory contains all editable files for your project—Java source files, editable descriptor files, JSPs, static content, and so forth. You create the source directory for an application by following the directory structure guidelines described in Organizing Jakarta EE Components in a Split Development Directory.

The top level of the source directory always represents an enterprise application (`.ear` file), even if you are developing only a single Jakarta EE module. Subdirectories beneath the top level source directory contain:

- Enterprise Application Modules (EJBs and Web applications)

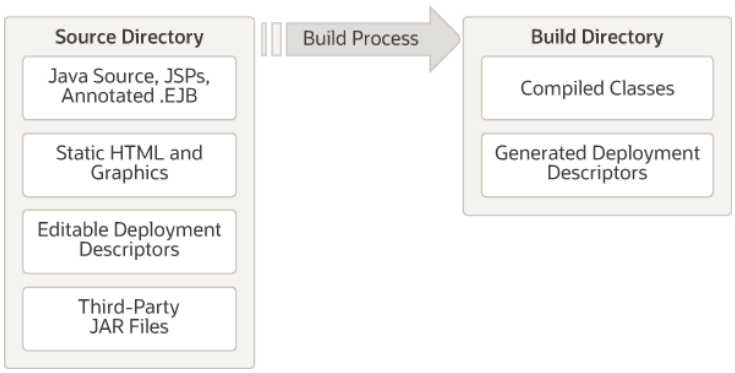
Copyright © 2007, 2026, Oracle and/or its affiliates.

> **Note**
>
> The split development directory structure does not provide support for developing new Resource Adapter components.

- Descriptor files for the enterprise application (`application.xml` and `weblogic-application.xml`)
- Utility classes shared by modules of the application (for example, exceptions, constants)
- Libraries (compiled `.jar` files, including third-party libraries) used by modules of the application

The build directory contents are generated automatically when you run the `wlcompile` ant task against a valid source directory. The `wlcompile` task recognizes EJB, Web application, and shared library and class directories in the source directory, and builds those components in an order that supports common class path requirements. Additional Ant tasks can be used to build Web services or generate deployment descriptor files from annotated EJB code.

Figure 4-1 Source and Build Directories



The build directory contains only those files generated during the build process. The combination of files in the source and build directories form a deployable Jakarta EE application.

The build and source directory contents can be placed in any directory of your choice. However, for ease of use, the directories are commonly placed in directories named `source` and `build`, within a single project directory (for example, `\myproject\build` and `\myproject\source`).

Deploying from a Split Development Directory

All WebLogic Server deployment tools (`weblogic.Deployer`, `wldeploy`, and the WebLogic Remote Console) support direct deployment from a split development directory. You specify only the build directory when deploying the application to WebLogic Server.

Direct split-directory deployment requires the server process to be able to read both the build directory and the source directory identified by .beabuild.txt. These paths are resolved in the server's filesystem. If the source tree is unavailable there, use wlpackage to create an EAR or a complete exploded EAR and deploy that packaged output.

WebLogic Server attempts to use all classes and resources available in the *source* directory for deploying the application. If a required resource is not available in the source directory,

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

WebLogic Server then looks in the application's build directory for that resource. For example, if a deployment descriptor is generated during the build process, rather than stored with source code as an editable file, WebLogic Server obtains the generated file from the build directory.

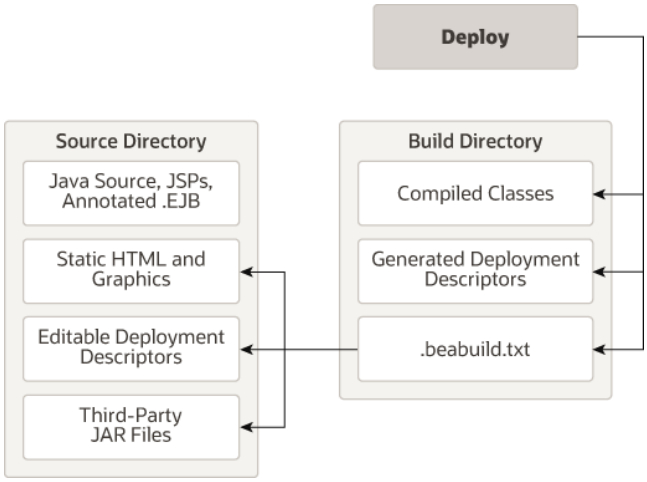
WebLogic Server discovers the location of the source directory by examining the .beabuild.txt file that resides in the top level of the application's build directory. After moving the source directory, update the build script's source path and run wlcompile again to regenerate .beabuild.txt. If you edit the file manually, set its bea.srcdir property to the new source-directory path; use Java-properties escaping, or forward slashes for a Windows path. A relative bea.srcdir value is resolved against the build directory.

```
bea.srcdir=C:/work/helloWorldEar/source
```

Deploying Applications Using wldeploy describes the `wldeploy` Ant task that you can use to automate deployment from the split directory environment.

Figure 4-2 shows a typical deployment process. The process is initiated by specifying the build directory with a WebLogic Server tool. In the figure, all compiled classes and generated deployment descriptors are discovered in the build directory, but other application resources (such as static files and editable deployment descriptors) are missing. WebLogic Server uses the hidden `.beabuild.txt` file to locate the application's source directory, where it finds the required resources.

Figure 4-2 Split Directory Deployment



Split Development Directory Ant Tasks

Oracle provides a collection of Ant tasks designed to help you develop applications using the split development directory environment. Each Ant task uses the source, build, or both directories to perform common development tasks:

- `wlcompile`—This Ant task compiles the contents of the source directory into subdirectories of the build directory. `wlcompile` compiles Java classes and also processes annotated `.ejb` files into deployment descriptors, as described in Compiling Applications Using wlcompile.
- `wlappc`—This Ant task invokes the appc compiler, which generates JSPs and container-specific EJB classes for deployment. See Building Modules and Applications Using wlappc.

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

- `wldeploy`—This Ant task deploys any format of Jakarta EE applications (exploded or archived) to WebLogic Server. To deploy directly from the split development directory environment, you specify the build directory of your application. See wldeploy Ant Task Reference.
- `wlpackage`—This Ant task uses the contents of both the source and build directories to generate an EAR file or exploded EAR directory that you can give to others for deployment.

Using the Split Development Directory Structure: Main Steps

In a split development directory structure, you can develop and deploy applications faster, simplify build scripts, and integrate with source control systems.

The following steps illustrate how you use the split development directory structure to build and deploy a WebLogic Server application.

1. Create the main EAR source directory for your project. When using the split development directory environment, you must develop Web applications and EJBs as part of an enterprise application, even if you do not intend to develop multiple Jakarta EE modules. See Organizing Jakarta EE Components in a Split Development Directory.
2. Add one or more subdirectories to the EAR directory for storing the source for Web applications, EJB components, or shared utility classes. See Organizing Jakarta EE Components in a Split Development Directory and Organizing Shared Classes in a Split Development Directory.
3. Store all of your editable files (source code, static content, editable deployment descriptors) for modules in subdirectories of the EAR directory. Add the entire contents of the source directory to your source control system, if applicable.
4. Set your WebLogic Server environment by executing either the `setWLSEnv.cmd` (Windows) or `setWLSEnv.sh` (UNIX) script. The scripts are located in the *`WL_HOME`*`\server\bin\` directory, 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.

5. Use the `weblogic.BuildXMLGen` utility to generate a default `build.xml` file for use with your project. Edit the default property values as needed for your environment. See Generating a Basic build.xml File Using weblogic.BuildXMLGen.
6. Use the default targets in the `build.xml` file to build, deploy, and package your application. See Generating a Basic build.xml File Using weblogic.BuildXMLGen for a list of default targets.

Organizing Jakarta EE Components in a Split Development Directory

The split development directory structure requires each project to be staged as a Jakarta EE enterprise application. Oracle therefore recommends that you stage even standalone Web applications and EJBs as modules of an enterprise application, to benefit from the split

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

directory Ant tasks. This practice also allows you to easily add or remove modules at a later date, because the application is already organized as an EAR.

> **Note**
>
> If your project requires multiple EARs, see also Developing Multiple-EAR Projects Using the Split Development Directory.

The following sections describe the basic conventions for staging the following module types in the split development directory structure:

- Enterprise Application Configuration
- Web Applications
- EJBs
- Shared Utility Classes
- Third-Party Libraries

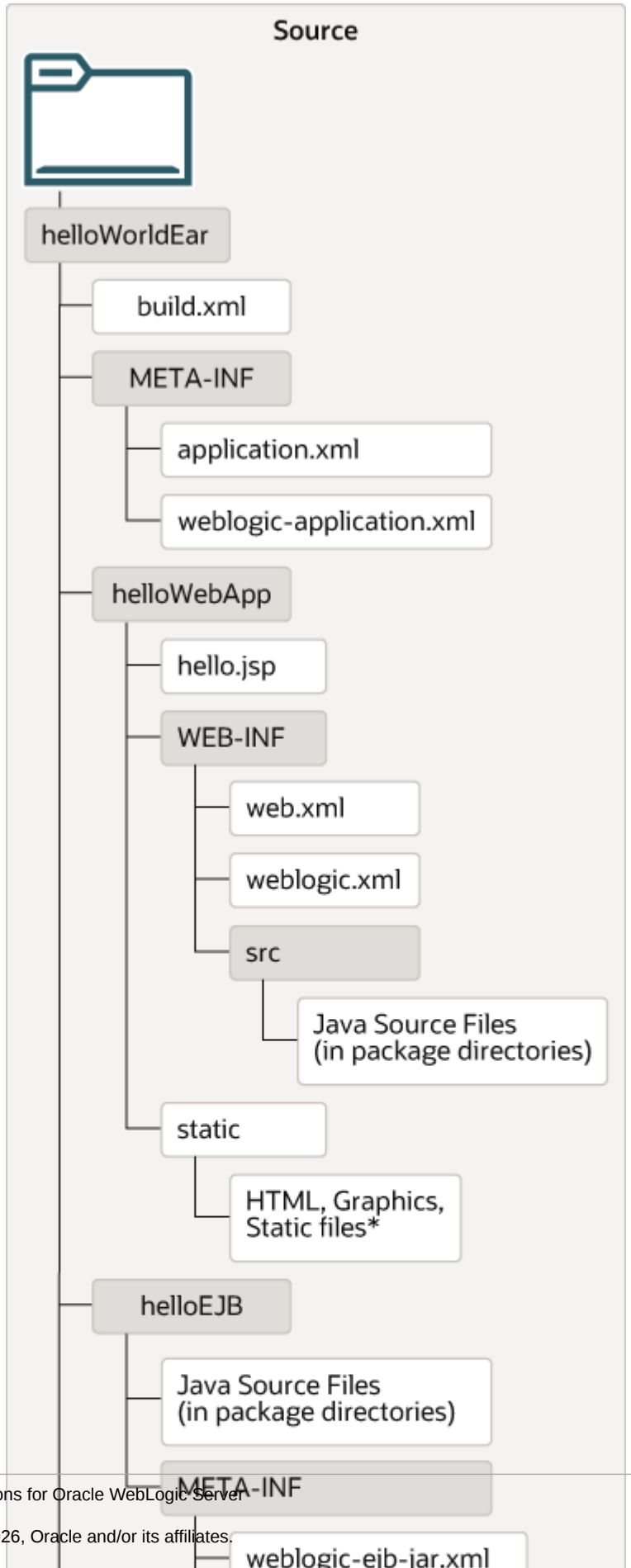
The directory examples are taken from the `splitdir` sample application installed in *`ORACLE_HOME`*`\wlserver\samples\server\examples\src\examples\splitdir`, where *`ORACLE_HOME`* represents the directory you specified as the Oracle Home when you installed WebLogic Server; for example, `C:\Oracle\Middleware\Oracle_Home`. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server.*

Source Directory Overview

The following figure summarizes the source directory contents of an enterprise application having a Web application, EJB, shared utility classes, and third-party libraries. The sections that follow provide more details about how individual parts of the enterprise source directory are organized.

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

Figure 4-3 Overview of Enterprise Application Source Directory

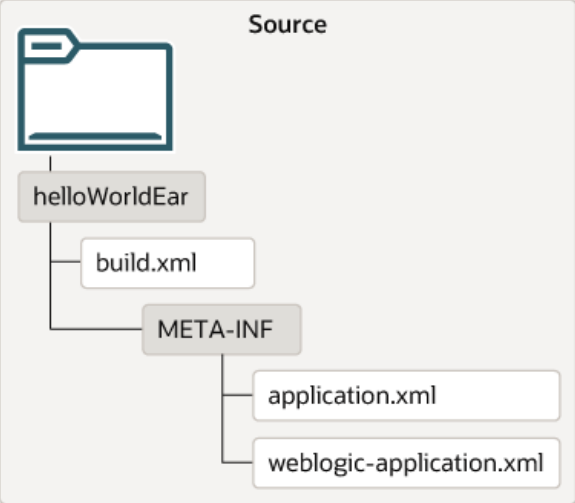


Copyright © 2007, 2026, Oracle and/or its affiliates.

Enterprise Application Configuration

The following figure shows the top-level layout used by the helloWorldEar sample. It includes META-INF/application.xml and META-INF/weblogic-application.xml. Include the descriptors required by your application's metadata and WebLogic-specific configuration; both files are not mandatory for every application. A generated build.xml can be stored here or in the output directory selected with weblogic.BuildXMLGen -d.

Figure 4-4 Enterprise Application Source Directory



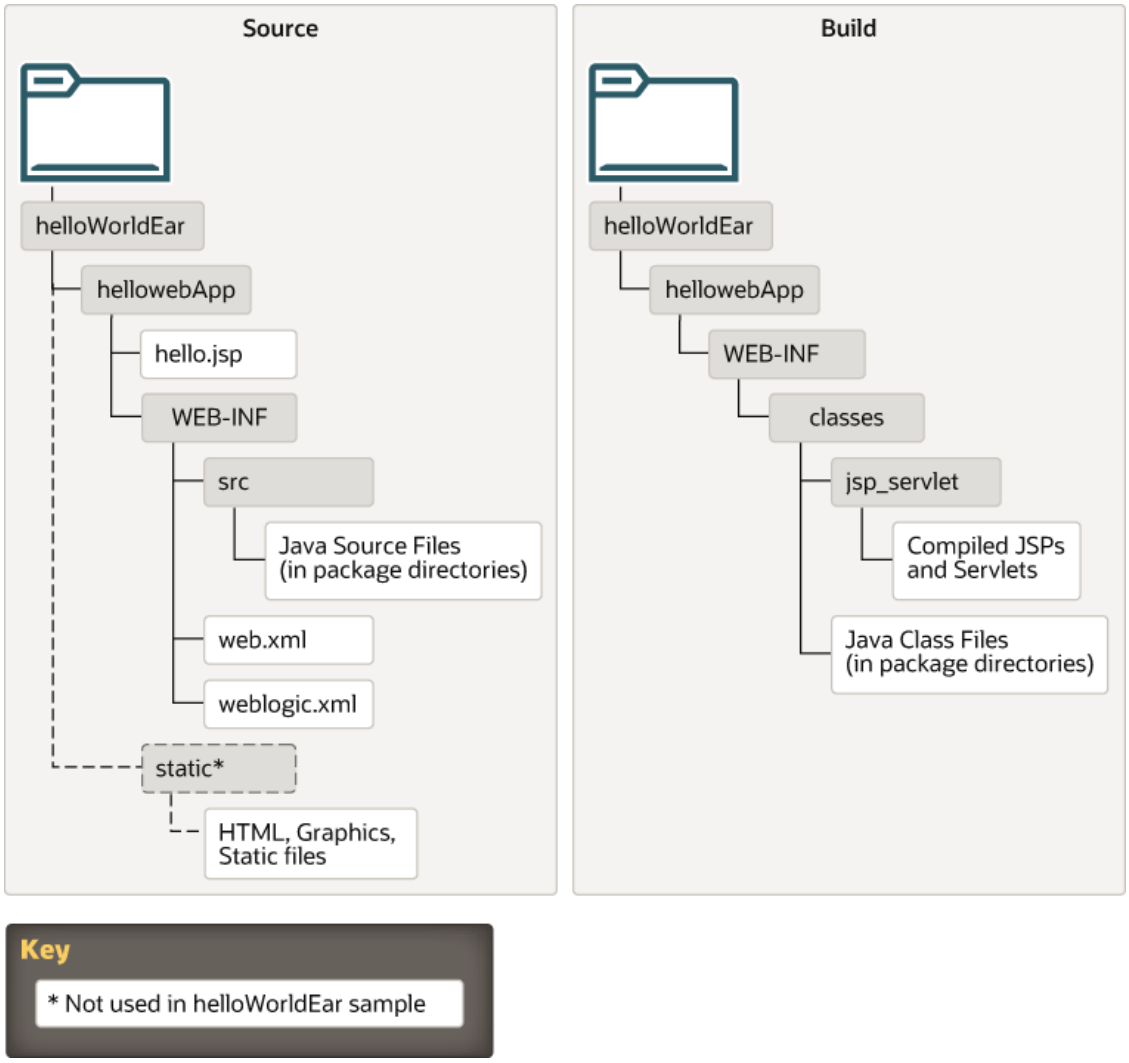
The enterprise application directory will also have one or more subdirectories to hold a Web application, EJB, utility class, and/or third-party Jar file, as described in the following sections.

Web Applications

Web applications use the basic source directory layout shown in the figure below.

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

Figure 4-5 Web Application Source and Build Directories



The key directories and files for the Web application are:

- `helloWebApp\` —The top level of the Web application module can contain JSP files and static content such as HTML files and graphics used in the application. You can also store static files in any named subdirectory of the Web application (for example, `helloWebApp\graphics` or `helloWebApp\static`.)
- `helloWebApp\WEB-INF\` —Store the Web application's editable deployment descriptor files (`web.xml` and `weblogic.xml`) in the `WEB-INF` subdirectory.
- `helloWebApp\WEB-INF\src` —Store Java source files for Servlets in package subdirectories under `WEB-INF\src`.

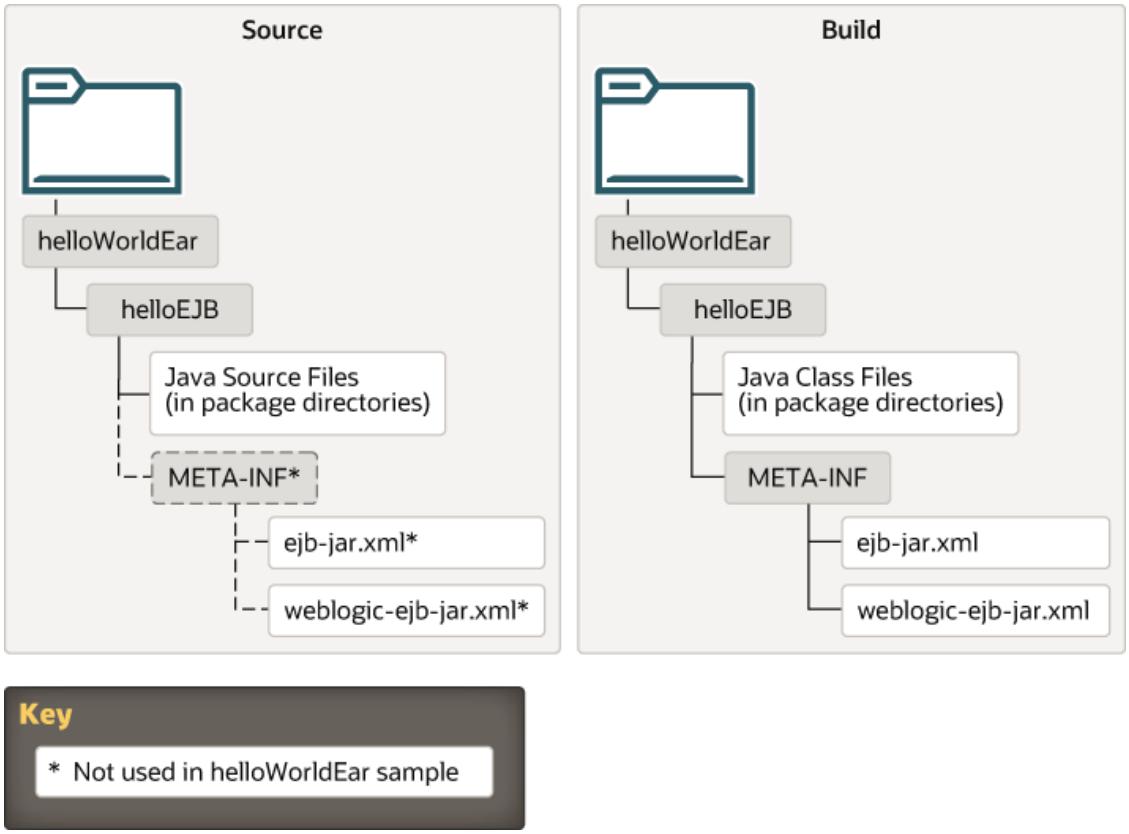
The wlcompile task compiles Java sources under helloWebApp/WEB-INF/src into package directories under helloWebApp/WEB-INF/classes in the build tree. To precompile JSPs, run the wlappc task against the compiled application. JSP-generated classes are separate from the application's Java-source classes. Editable Web deployment descriptors remain in the source tree unless a custom build step copies them. The generated JSP package layout shown in Figure 4-5 is illustrative; JSP-generated classes and classes compiled from WEB-INF/src are distinct outputs.

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

EJBs

EJBs use the source directory layout shown in the figure below.

Figure 4-6 EJB Source and Build Directories



The key directories and files for an EJB are:

- helloEJB\ —Store all EJB source files under package directories of the EJB module directory. Use .java source files; these may contain Jakarta Enterprise Beans annotations.
- helloEJB/META-INF/—Store the EJB module's editable deployment descriptors here. The helloWorldEar sample includes both ejb-jar.xml and weblogic-ejb-jar.xml in this directory. Its build script copies these descriptors to helloEJB/META-INF in the build tree; wlcompile itself compiles the Java sources. See Important Notes Regarding EJB Descriptors.

  Figure 4-6 retains an older sample layout. In the current helloWorldEar sample, both EJB descriptors are maintained in the source tree and copied by the sample build script.

During the build process, EJB classes are compiled into package subdirectories of the helloEJB module in the build directory.

Important Notes Regarding EJB Descriptors

Keep manually maintained EJB deployment descriptors in the source module's META-INF directory. Jakarta Enterprise Beans annotations in .java files can define metadata without an ejb-jar.xml descriptor when the specification permits it; deployment descriptors can supplement or override annotation metadata. wlcompile compiles Java sources and does not generate

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

descriptors from .ejb files. If a separate generation step is part of your own build, keep its generated descriptors in the build tree and do not maintain a second conflicting copy as source.

Organizing Shared Classes in a Split Development Directory

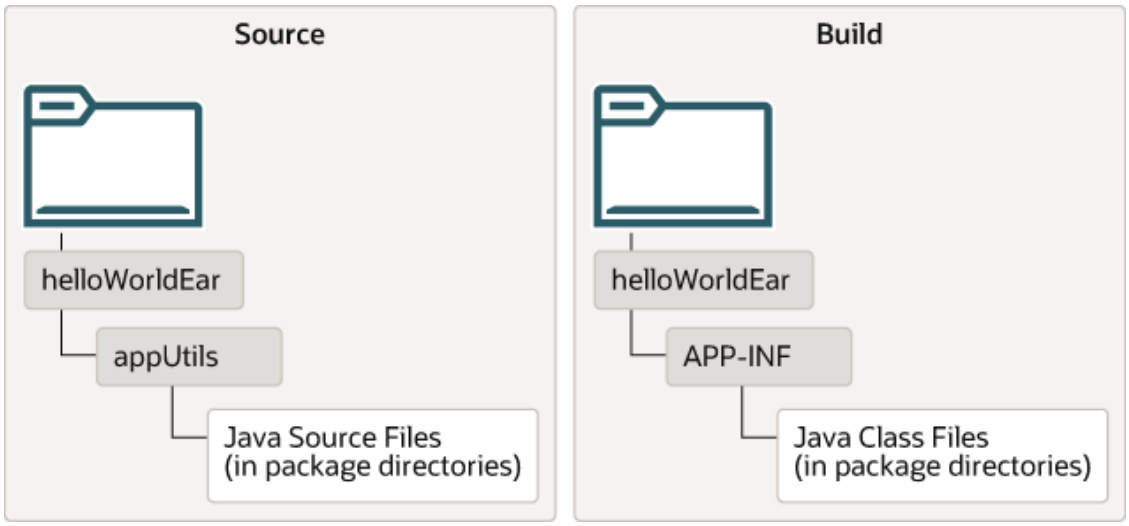
The WebLogic split development directory also helps you store shared utility classes and libraries that are required by modules in your enterprise application.

The following sections describe the directory layout and classloading behavior for shared utility classes and third-party JAR files.

Shared Utility Classes

Enterprise applications frequently use Java utility classes that are shared among application modules. Java utility classes differ from third-party JARs in that the source files are part of the application and must be compiled. Java utility classes are typically libraries used by application modules such as EJBs or Web applications.

Figure 4-7 Java Utility Class Directory



Place the source for Java utility classes in a named subdirectory of the top-level enterprise application directory. Beneath the named subdirectory, use standard package subdirectory conventions.

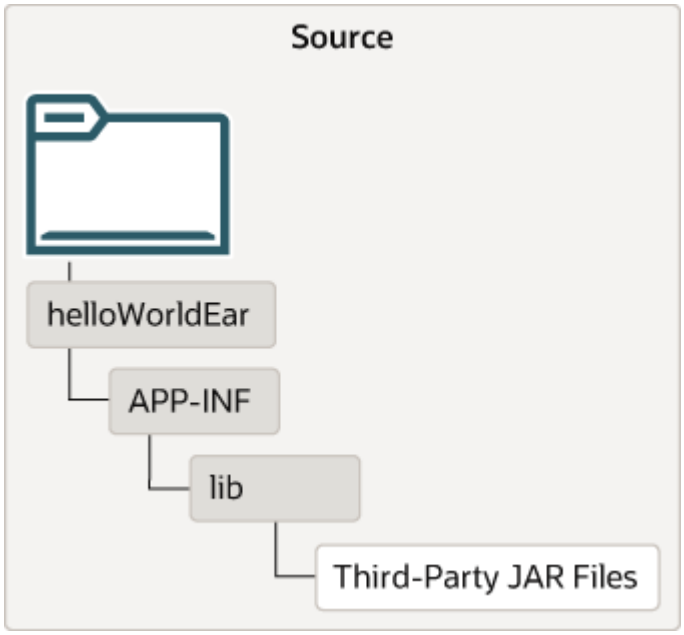
During the build process, the `wlcompile` Ant task invokes the javac compiler and compiles Java classes into the `APP-INF/classes/` directory under the build directory. This ensures that the classes are available to other modules in the deployed application.

Third-Party Libraries

You can extend an enterprise application to use third-party `.jar` files by placing the files in the `APP-INF\lib\` directory, as shown below:

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

Figure 4-8 Third-party Library Directory



Third-party JARs are generally not compiled, but may be versioned using the source control system for your application code. For example, XML parsers, logging implementations, and Web application framework JAR files are commonly used in applications and maintained along with editable source code.

During the build process, third-party JAR files are not copied to the build directory, but remain in the source directory for deployment.

Class Loading for Shared Classes

The classes and libraries stored under APP-INF/classes and APP-INF/lib are available to all modules in the enterprise application. When searching the application's own classpath, the application classloader searches APP-INF/classes before JARs in APP-INF/lib. Class resolution also follows the classloader delegation rules and configured package preferences; this ordering does not guarantee that an application class overrides a class visible from a parent classloader. See Understanding WebLogic Server Application Classloading.
Understanding WebLogic Server Application Classloading

Generating a Basic build.xml File Using weblogic.BuildXMLGen

After you set up your source directory structure, use the `weblogic.BuildXMLGen` utility to create a basic `build.xml` file. `weblogic.BuildXMLGen` is a convenient utility that generates an Ant build.xml file for enterprise applications that are organized in the split development directory structure. The utility analyzes the source directory and creates build and deploy targets for the enterprise application as well as individual modules. It also creates a target to clean the build directory.

BuildXMLGen scans application and module manifests for optional-package references and attempts to resolve them against libraries registered from -librarydir. Resolved references are added as nested library elements in generated wlcompile and wlappc tasks. Put the referenced

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

library in a library search directory and pass that directory when generating the build file. An unresolved reference is not converted into a usable library element.

For example, when lib/echolib.jar is a resolvable optional package, a generated task includes a library element for that JAR. This illustrative path is relative to the generated build file's project directory.

```
<target name="appc" description="Runs weblogic.appc on your application">
  <wlappc source="${dest.dir}" verbose="${verbose}">
    <library file="lib/echolib.jar"/>
  </wlappc>
</target>
```

The generated wlcompile and applicable module targets use the same resolved library.

Substitute actual absolute paths for the illustrative /work paths. Choose an output directory that does not already contain the selected build filename:

```
java weblogic.BuildXMLGen -librarydir "/work/helloWorldEar/lib" -d "/work/
helloWorldEar" "/work/helloWorldEar/source"
```

weblogic.BuildXMLGen Syntax

The syntax for `weblogic.BuildXMLGen` is as follows:

```
java weblogic.BuildXMLGen [options] <source directory>
```

where `options` include:

- `-help`—Print standard usage message.
- `-version`—Print version information.
- `-projectName <project name>`—Name of the Ant project.
- `-d <directory>`—Directory where `build.xml` is created. The default is the current directory.
- `-file <build.xml>`—Name of the generated build file.
- `-librarydir <directories>`—Comma-separated list of directories in which to register and search for shared libraries and optional packages used by application references. Generated library targets correspond to the references found, not every JAR in each directory.
- `-username <username>`—User name for deploy commands.

  These values are written into the generated build file. For reusable or source-controlled build files, use the stored-credential customization below instead of supplying or committing a password.
- `-password <password>`—User password.

  These values are written into the generated build file. For reusable or source-controlled build files, use the stored-credential customization below instead of supplying or committing a password.
- `-adminurl <url>`
  —Administration Server URL written to the generated adminurl property; default iiop://localhost:7001. Set this to the endpoint used by your development domain.

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

After running `weblogic.BuildXMLGen`, edit the generated `build.xml` file to specify properties for your development environment. The list of properties you need to edit are shown in the listing below:

Example 4-1 Properties to Customize After Generation

```
<!-- BUILD PROPERTIES ADJUST THESE FOR YOUR ENVIRONMENT -->
<property name="tmp.dir" value="/tmp" />
<property name="dist.dir" value="${tmp.dir}/dist"/>
<property name="app.name" value="helloWorldEar" />
<property name="ear" value="${dist.dir}/${app.name}.ear"/>
<property name="ear.exploded" value="${dist.dir}/${app.name}_exploded"/>
<property name="verbose" value="true" />
<property name="servername" value="myserver" />
<property name="adminurl" value="iiop://localhost:7001" />
```

Choose tmp.dir as a writable parent directory for generated files. The generated dest.dir is ${tmp.dir}/${app.name}; dist.dir defaults to ${tmp.dir}/dist. The -d option chooses where the Ant build file is written and does not choose dest.dir. src.dir identifies the source tree; use an absolute source-directory argument when generating a build file in another directory.

The following listing shows selected common targets in the generated build file. Run ant -f <generated-build-file> -projecthelp to list the targets in that file.

Run this example from a project root containing source/. The ant output directory must not already contain build.xml.</p> <codeblock outputclass="language-sh">mkdir -p ant java weblogic.BuildXMLGen -projectName helloWorldEar -d "$PWD/ant" "$PWD/source" ant -f "$PWD/ant/build.xml" -projecthelp

The absolute file paths are illustrative; substitute the actual protected user configuration and key file paths. servername must match the target server used by module-redeployment targets.

"language-sh">ant -f /work/helloWorldEar/ant/build.xml \ -Dwls.userConfigFile=/private/credentials/wls-user.config \ -Dwls.userKeyFile=/private/credentials/wls-user.key deploy

"language-xml"><wldeploy action="deploy" source="${dest.dir}" name="${app.name}" adminurl="${adminurl}" userConfigFile="${wls.userConfigFile}" userKeyFile="${wls.userKeyFile}"/>

Create a WebLogic user-configuration file and its matching key file using storeUserConfig as described with the stored-credential examples in Chapter 2. Keep the pair outside source control and restrict access. In every generated wldeploy task, including deploy, undeploy, module-redeploy and any library targets, remove both user and password attributes and add userConfigFile="${wls.userConfigFile}" and userKeyFile="${wls.userKeyFile}". Remove the unused user/password property declarations as well. Supply the two file paths when invoking a deployment target.

Example: Common Targets in a Generated build.xml
appc Runs weblogic.appc on your application build Compiles helloWorldEar application and runs appc clean Deletes the build directory compile Only compiles helloWorldEar application, no appc deploy Deploys (and redeploys) the entire helloWorldEar application ear Package a standard Java EE EAR for distribution ear.exploded Package a standard exploded Java EE EAR config.server Configure server with resources required by application undeploy Undeploys the entire helloWorldEar application

This is a selected list. The generator also emits compile.<directory> and redeploy.<directory> targets for top-level source directories other than META-INF and APP-INF. For this sample the source directories are appStartup, appUtils, helloEJB and helloWebApp. Use module-redeployment targets only for actual deployable modules, such as helloEJB or helloWebApp; utility-source directories do not become Jakarta EE modules because a target exists. Keep

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

unrelated directories outside the source root. The config.server target is a customization placeholder and does not configure server resources until you implement it. The clean target leaves distribution archives in dist.dir; remove those separately when needed.

Developing Multiple-EAR Projects Using the Split Development Directory

The preceding examples use a single enterprise application. A project that builds multiple EARs can share library and utility-class inputs and coordinate the EAR build files, as described in the following sections.

> **Note**
>
> The retained MedRec diagram is a historical illustration of the multiple-EAR sharing pattern. Its directory and JAR names are not the installed sample layout for this release. Consult the installed sample's own instructions for its current source location and build commands.

Organizing Libraries and Classes Shared by Multiple EARs

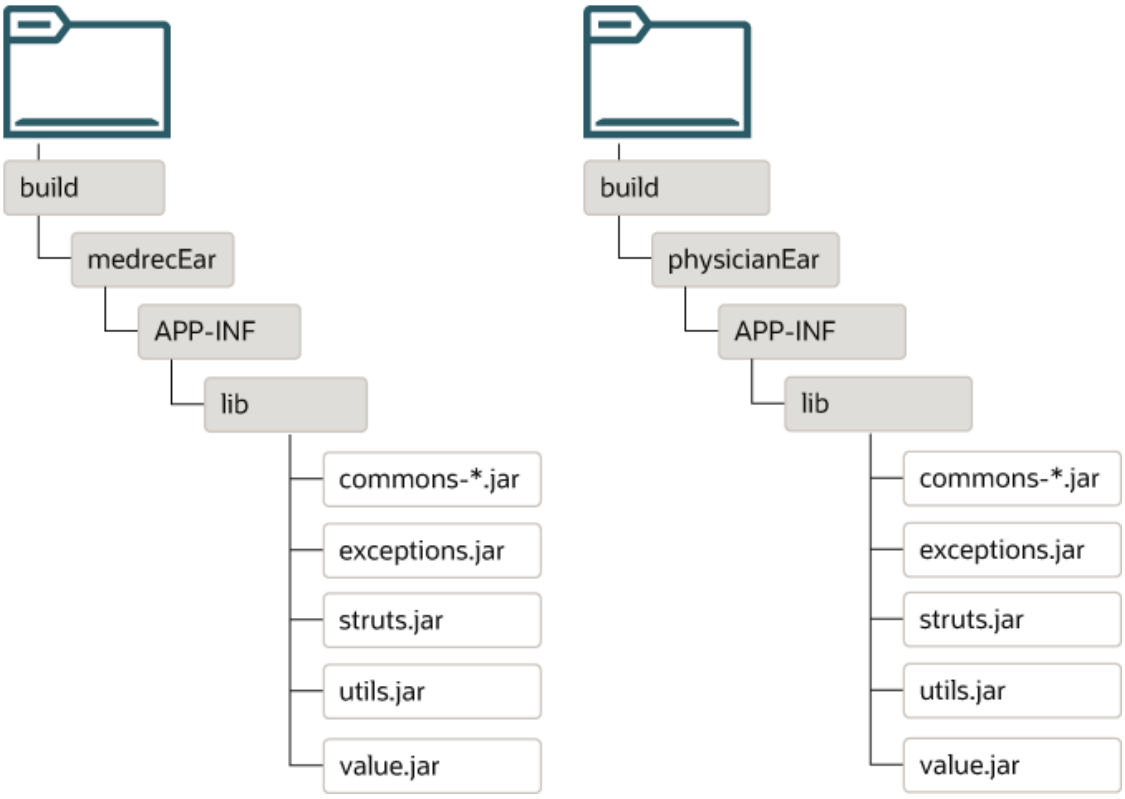
For single EAR projects, the split development directory conventions suggest keeping third-party JAR files in the `APP-INF/lib` directory of the EAR source directory. However, a multiple-EAR project would require you to maintain a copy of the same third-party JAR files in the `APP-INF/lib` directory of *each* EAR source directory. This introduces multiple copies of the source JAR files, increases the possibility of some JAR files being at different versions, and requires additional space in your source control system.

To address these problems, consider editing your build script to copy third-party JAR files into the `APP-INF/lib` directory of the *build* directory for each EAR that requires the libraries. This allows you to maintain a single copy and version of the JAR files in your source control system, yet it enables each EAR in your project to use the JAR files.

The MedRec sample application installed with WebLogic Server uses this strategy, as shown in the following figure.

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

Figure 4-9 Shared JAR Files in MedRec



MedRec takes a similar approach to utility classes that are shared by multiple EARs in the project. Instead of including the source for utility classes within the scope of each ear that needs them, MedRec keeps the utility class source independent of all EARs. After compiling the utility classes, the build script archives them and copies the JARs into the build directory under the `APP-INF/LIB` subdirectory of each EAR that uses the classes, as shown in figure Figure 4-9.

Linking Multiple build.xml Files

When developing multiple EARs using the split development directory, each EAR project generally uses its own `build.xml` file (perhaps generated by multiple runs of `weblogic.BuildXMLGen`.). Applications like MedRec also use a main `build.xml` file that calls the other `build.xml` files for each EAR in the application suite.

Ant provides a core task (named `ant`) that allows you to execute other project build files within a main `build.xml` file. The following line from the MedRec main build file shows its usage:

```
<ant inheritAll="false" dir="${root}/startupEar" antfile="build.xml"/>
```

The above task instructs Ant to execute the file named `build.xml` in the `/startupEar` subdirectory. The `inheritAll` parameter instructs Ant to pass only user properties from the main build file to the `build.xml` file in `/startupEar`.

MedRec uses multiple tasks similar to the above to build the `startupEar`, `medrecEar`, and `physicianEar` applications, as well as building common utility classes and client applications.

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

Best Practices for Developing WebLogic Server Applications

The WebLogic Server documentation library includes a number of recommended best practices for application development, including topics such as packaging, distribution, deployment, and more.

For the Ant split development directory workflow described in this chapter, use the following practices. Other supported build workflows, including Maven, can use their own source layouts.

- Organize this split-directory project as an enterprise application, including when it contains only one Web or EJB module. See Organizing Jakarta EE Components in a Split Development Directory.Organizing Jakarta EE Components in a Split Development Directory
- Use the source/build separation and the corresponding split-directory Ant tasks described in this chapter.
- For distribution purposes, package and deploy in archived format. See Packaging Applications Using wlpackage.
- In most other cases, it is more convenient to deploy in exploded format. See Archive versus Exploded Archive Directory.
- Never deploy untested code on a WebLogic Server instance that is serving production applications. Instead, set up a development WebLogic Server instance on the same computer on which you edit and compile, or designate a WebLogic Server development location elsewhere on the network.
- Set the compile-time classpath to the APIs used by your application. Applications using only standard Jakarta EE 11 APIs can compile against the Jakarta EE 11 API dependency without installing WebLogic Server on the compilation machine. In Maven, use jakarta.platform:jakarta.jakartaee-api:11.0.0 with provided scope; the target server supplies these APIs at runtime. Do not bundle the platform API JAR in the application. WebLogic-specific APIs and the WebLogic Ant tools used in this chapter require the corresponding WebLogic Server distribution classes and environment.

Using the WebLogic Maven Plug-In

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

Optional Maven compile-time API dependency:

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

5

Building Applications in a Split Development Directory

To build WebLogic Server Jakarta EE applications in WebLogic split development directory environment you have to compile applications using `wlcompile` and build modules and applications using `wlappc`.

Compiling Applications Using wlcompile

You can use the `wlcompile` Ant task to invoke the javac compiler to compile your application's Java components in a split development directory structure.

The basic syntax of `wlcompile` identifies the source and build directories, as in this command from the `helloWorldEar` sample:

```
<wlcompile srcdir="${src.dir}" destdir="${dest.dir}"/>
```

> **Note**
>
> Deployment descriptors are optional for applications whose metadata can be supplied by annotations or defaults. For wlcompile, a Web module is recognized by its WEB-INF directory or by Web-archive recognition, including a .war directory suffix. An EJB module is recognized from its EJB descriptor or annotated Java sources; a .jar directory suffix also identifies an EJB module. Use a .war or .jar suffix when the directory's contents do not otherwise identify its module type. These suffixes identify exploded module directories; they do not turn utility-source directories into deployable modules.

The following is the order in which events occur using this task:

1. First, wlcompile compiles shared Java utility sources into APP-INF/classes under the build directory selected by destdir. In the preceding task fragment, that output is ${dest.dir}/APP-INF/classes. The helloWorldEar sample defines dest.dir from its examples build settings; it is not a fixed installation path. For the installed sample's locations and configuration, see Sample Applications and Code Examples in Understanding Oracle WebLogic Server. Sample Applications and Code Examples
2. `wlcompile` builds the EJBs and automatically includes the previously built Java modules in the compiler's classpath. This allows the EJBs to call the Java modules without requiring you to manually edit their classpath.
3. Finally, `wlcompile` compiles the Java components in the Web application with the EJB and Java modules in the compiler's classpath. This allows the Web applications to refer to the EJB and application Java classes without requiring you to manually edit the classpath.

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

Using includes and excludes Properties

Use the includes and excludes attributes to select which immediate subdirectories of the enterprise application source root participate in a wlcompile invocation. Values are Ant include/exclude patterns relative to srcdir. These attributes select directories; they do not establish an arbitrary dependency order. For dependencies that need separate compilation stages, invoke wlcompile separately with the appropriate directory selections in the required order.

The following line from the helloWorldEar sample excludes the appStartup utility-source directory from this compilation invocation:

```
<wlcompile srcdir="${src.dir}" destdir="${dest.dir}"
   excludes="appStartup"/>
```

wlcompile Ant Task Attributes

Table 5-1 contains Ant task attributes specific to `wlcompile`.

Table 5-1 wlcompile Ant Task Attributes

Attribute	Description
`srcdir`	The source directory.
`destdir`	The build/output directory.
`classpath`	Appends entries to the compilation classpath initialized from the Ant process/task environment; does not replace that existing classpath.
`includes`	Ant patterns selecting immediate source-root directories to compile.
`excludes`	Ant patterns excluding immediate source-root directories from this invocation.
`librarydir`	Specifies a directory of shared Java EE libraries to add to the classpath. See Creating Shared Java EE Libraries and Optional Packages.

Nested javac Options

The `wlcompile` Ant task can accept nested javac options to change the compile-time behavior. For example, the following `wlcompile` command ignores deprecation warnings and enables debugging:

```
<wlcompile srcdir="${mysrcdir}" destdir="${mybuilddir}">
   <javac deprecation="false" debug="true"
   debuglevel="lines,vars,source"/>
</wlcompile>
```

Setting the Classpath for Compiling Code

Run the WebLogic split-directory Ant tasks after setting the WebLogic environment as described in Chapter 4. The tasks require the matching WebLogic distribution classes; wlcompile adds the module and shared-class paths described above. For application code that uses WebLogic public APIs, the distribution provides WL_HOME/server/lib/wls-api.jar, where WL_HOME is the WebLogic Server installation directory. Add any third-party libraries and other application classes required by your source. Applications using only standard Jakarta EE APIs can instead use the corresponding standard API dependencies, as described in Chapter 3 and the Chapter 4 compilation guidance. Do not add a JDK lib/tools.jar: that file was removed with

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

the modular JDK layout. For installed samples, load examples.properties through the sample build script's property task; do not add the properties file itself to the compiler classpath. Follow the installed sample's instructions for its locations and build properties.

Related build workflows:

Using the WebLogic Maven Plug-In
;

Creating a Split Development Directory Environment

Library Element for wlcompile and wlappc

Use an optional nested library element in wlcompile or wlappc to supply an individual shared-library file to the build tool. The file attribute is required. The optional name, specificationversion and implementationversion attributes identify the supplied library's registration metadata; they do not themselves declare an application dependency or a minimum required version. Declare application library references in the appropriate deployment descriptor, or optional-package references in the manifest, as described in Creating Shared Jakarta EE Libraries and Optional Packages.

Table 5-2 Library attributes

Attribute	Description
`file`	Path to the supplied library archive or exploded library directory; required and must exist.
`name`	Optional registration name of the supplied library.
`specificationversion`	Optional specification version of the supplied library.
`implementationversion`	Optional implementation version of the supplied library.

The format choices for both specificationversion and implementationversion are described in Referencing Shared Jakarta EE Libraries in an Enterprise Application. The following configuration fragment supplies a library file with registration metadata; use values appropriate to that library:

```
<library file="c:\mylibs\lib.jar" name="ReqLib" specificationversion="920"
implementationversion="1.1" />
```

Building Modules and Applications Using wlappc

Use weblogic.appc, or the wlappc Ant task, to precompile JSPs, generate required EJB support classes and perform deployment-descriptor and cross-module validation. The input can be a supported application/module archive or exploded directory. These build-time checks do not establish every aspect of Jakarta EE application compliance or replace deployment and runtime testing.

The application-level checks include checks between the application-level deployment descriptors and the individual modules, as well as validation checks across the modules.

Additionally, optional packages are supported as Jakarta EE shared libraries in `appc`, whereby all manifests of an application and its modules are scanned to look for optional package references.

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

wlappc is the Ant task interface to the weblogic.appc compiler. The following sections describe the wlappc options and usage. For an EAR with application.xml, appc processes its standard modules in descriptor order. For a descriptor-free EAR, module discovery supplies the application descriptor model used by the compiler. This ordering statement concerns appc; wlcompile uses the utility/EJB/Web compilation order described earlier.

wlappc Ant Task Attributes

Table 5-3 describes Ant task options specific to `wlappc`. These options are similar to the `weblogic.appc` command-line options, but with a few differences.

> **Note**
>
> See weblogic.appc Reference for a list of `weblogic.appc` options.
>
> See also Library Element for wlcompile and wlappc.

Table 5-3 wlappc Ant Task Attributes

Option	Description
version <level>	Sets the Java source and target levels passed for generated Java compilation, for example version="21". This is not the WebLogic product version. Use levels supported by the selected compiler and deployment runtime.
`output <file>`	Specifies an alternate output archive or directory. If not set, the output is placed in the source archive or directory.
`forceGeneration`	Forces generation of EJB and JSP classes. Without this flag, the classes may not be regenerated (if determined to be unnecessary).
`lineNumbers`	Adds line numbers to generated class files to aid in debugging.
`basicClientJar`	Does not include deployment descriptors in client JARs generated for EJBs.
`idl`	Generates IDL for EJB remote interfaces.
`idlOverwrite`	Always overwrites existing IDL files.
`idlVerbose`	Displays verbose information for IDL generation.
`idlNoValueTypes`	Does not generate valuetypes and the methods/attributes that contain them.
`idlNoAbstractInterfaces`	Does not generate abstract interfaces and methods/attributes that contain them.
`idlFactories`	Generates factory methods for valuetypes.
`idlVisibroker`	Generates IDL somewhat compatible with Visibroker 4.5 C++.
`idlOrbix`	Generates IDL somewhat compatible with Orbix 2000 2.0 C++.
`idlDirectory <dir>`	Specifies the directory where IDL files will be created (default: target directory or JAR)
idlMethodSignatures <signature>	Specifies the method signatures used to trigger IDL code generation.

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

Table 5-3 (Cont.) wlappc Ant Task Attributes

Option	Description
`iiop`	Generates CORBA stubs for EJBs.
`iiopDirectory <dir>`	Specifies the directory where IIOP stub files will be written (default: target directory or JAR)
`keepgenerated`	Keeps the generated `.java` files.
`librarydir`	Specifies a directory of shared Jakarta EE libraries to add to the classpath. See Creating Shared Jakarta EE Libraries and Optional Packages.
compiler <name>	Selects the Java compiler mode. Accepted values are javac, jdt and none. The default generated-EJB compiler mode is jdt.
`debug`	Compiles debugging information into a class file.
`optimize`	Compiles with optimization on.
`nowarn`	Compiles without warnings.
`verbose`	Compiles with verbose output.
`deprecation`	Warns about deprecated calls.
`normi`	Legacy compiler pass-through switch retained by the parser. It does not select a compiler; current appc compiler values are javac, jdt and none.
`runtimeflags`	Compiler-runtime pass-through arguments, where supported by the selected external Java compiler. This value is passed to appc's compiler argument parser; it does not configure the JVM running Ant or appc.
`classpath <path>`	Selects the classpath to use during compilation.
`clientJarOutputDir <dir>`	Specifies a directory to place generated client jar files. If not set, generated jar files are placed into the same directory location where the JVM is running.
source <path>	Required path to the application/module archive or exploded directory to process.
plan <file>	Optional deployment-plan XML file applied during compilation.
continueCompilation	Boolean; when true, continues JSP compilation after individual JSP errors. Default: false.

To configure Ant's JVM heap, use ANT_OPTS as described in Changing the Ant Heap Size. For the appc JVM launched directly with java, put JVM options before the weblogic.appc class name. Changing the Ant Heap Size

Command-line help, product-version output and inferred-descriptor writing are not exposed by these Ant attributes. Use the command-line options described in weblogic.appc Reference for those operations.

wlappc Ant Task Syntax

Set the required source attribute to the application or module archive/directory to process. In a split-directory build, ${dest.dir} is the build tree produced by wlcompile. If output is omitted, appc writes its output back to the input archive or directory.

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

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

The following task enables IDL generation and permits existing IDL output to be overwritten:

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

Syntax Differences between appc and wlappc

The command-line tool uses standalone switches for Boolean options, such as -idl. Ant expresses the corresponding option as an XML attribute with an explicit Boolean value, such as idl="true". Options that take paths or other values use -option value on the command line and option="value" in Ant. The following examples select the same existing input archive, foo.ear:

```
java weblogic.appc -idl foo.ear
```

```
<wlappc source="foo.ear" idl="true"/>
```

weblogic.appc Reference

The following sections describe how to use the command-line version of the appc compiler. The command-line compiler reports descriptor warnings and errors and compiles the relevant modules. It preserves the input archive/directory as its output by default; use -output to select another output archive or directory. Processing a standalone module does not automatically wrap it in an EAR.

weblogic.appc Syntax

Use the following syntax to run `appc`:

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

weblogic.appc Options

Table 5-4 lists selected appc options. Use -help for standard usage and -advanced for the extended option list available in this release.

Table 5-4 weblogic.appc Options

Option	Description
-help	Prints standard usage information.
`-version`	Prints `appc` version information.
`-output <file>`	Specifies an alternate output archive or directory. If not set, the output is placed in the source archive or directory.
`-forceGeneration`	Forces generation of EJB and JSP classes. Without this flag, the classes may not be regenerated (if determined to be unnecessary).
`-library <file[[@name=<string>] [@libspecver=<version>] [@libimplver=<version|string>]]>`	A comma-separated list of shared Java EE libraries. Optional name and version string information must be specified in the format described in Referencing Shared Java EE Libraries in an Enterprise Application.

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

Table 5-4 (Cont.) weblogic.appc Options

Option	Description
`-writeInferredDescriptors`	Writes deployment descriptors containing metadata inferred from annotations into the compilation output. To keep the input archive separate from that output, specify -output.
`-lineNumbers`	Adds line numbers to generated class files to aid in debugging.
`-basicClientJar`	Does not include deployment descriptors in client JARs generated for EJBs.
`-idl`	Generates IDL for EJB remote interfaces.
`-idlOverwrite`	Always overwrites existing IDL files.
`-idlVerbose`	Displays verbose information for IDL generation.
`-idlNoValueTypes`	Does not generate valuetypes and the methods/attributes that contain them.
`-idlNoAbstractInterfaces`	Does not generate abstract interfaces and methods/attributes that contain them.
`-idlFactories`	Generates factory methods for valuetypes.
`-idlVisibroker`	Generates IDL somewhat compatible with Visibroker 4.5 C++.
`-idlOrbix`	Generates IDL somewhat compatible with Orbix 2000 2.0 C++.
`-idlDirectory <dir>`	Specifies the directory where IDL files will be created (default: target directory or JAR)
-idlMethodSignatures <signature>	Specifies the method signatures used to trigger IDL code generation.
`-iiop`	Generates CORBA stubs for EJBs.
`-iiopDirectory <dir>`	Specifies the directory where IIOP stub files will be written (default: target directory or JAR)
`-keepgenerated`	Keeps the generated `.java` files.
-compiler <name>	Selects the Java compiler mode. Accepted values are javac, jdt and none. The default generated-EJB compiler mode is jdt.
`-g`	Compiles debugging information into a class file.
`-O`	Compiles with optimization on.
`-nowarn`	Compiles without warnings.
`-verbose`	Compiles with verbose output.
`-deprecation`	Warns about deprecated calls.
`-normi`	Legacy compiler pass-through switch retained by the parser. It does not select a compiler; current appc compiler values are javac, jdt and none.
`-J<option>`	Passes an option to the runtime of an externally invoked Java compiler. The inline compiler path ignores -J options. This does not configure the JVM running weblogic.appc.
`-classpath <path>`	Selects the classpath to use during compilation.
`-clientJarOutputDir <dir>`	Specifies a directory to place generated client jar files. If not set, generated jar files are placed into the same directory location where the JVM is running.
`-advanced`	Prints advanced usage options.

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

To configure Ant's JVM heap, use ANT_OPTS as described in Changing the Ant Heap Size. For the appc JVM launched directly with java, put JVM options before the weblogic.appc class name. Changing the Ant Heap Size

```
java weblogic.appc -writeInferredDescriptors -output application-inferred.ear
application.ear
```

For an existing application.ear, write inferred descriptors into a separate output archive:

```
java weblogic.appc -help
java weblogic.appc -advanced
```

The following commands display usage without compiling an application:

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

6

Deploying and Packaging from a Split Development Directory

To deploy and package WebLogic Server Jakarta EE applications in WebLogic split development directory environment use `wldeploy` and `wlpackage` tasks.

Deploying Applications Using wldeploy

The wldeploy task provides an easy way to deploy directly from the split development directory. The wldeploy Ant task exposes many of the options of the weblogic.Deployer command-line utility.

For direct deployment from a split development directory, set source to the build directory produced by wlcompile. The server process must be able to read that build tree and the source directory identified by its .beabuild.txt file; the paths are resolved in the server's filesystem. If the source tree is unavailable there, use wlpackage to create a complete EAR or exploded EAR and deploy that packaged output instead.

This example uses the stored-credential customization described in Chapter 4. Set dest.dir to the split build tree, app.name to the deployment name, adminurl to the intended Administration Server endpoint, and wls.userConfigFile/wls.userKeyFile to the protected matching credential files already created for that account.

```
<wldeploy action="deploy" source="${dest.dir}" name="${app.name}"
          adminurl="${adminurl}"
          userConfigFile="${wls.userConfigFile}"
          userKeyFile="${wls.userKeyFile}"/>
```

BuildXMLGen generates a deploy target. Customize that target's credential attributes as described in Generating a Basic build.xml File Using weblogic.BuildXMLGen; the preceding task shows the resulting stored-credential form.

See wldeploy Ant Task Reference, for a complete command reference.

Packaging Applications Using wlpackage

Use `wlpackage` when you want to deliver your application to another group or individual for evaluation, testing, performance profiling, or production deployment.

The `wlpackage` Ant task uses the contents of both the source and build directories to create either a deployable archive file (`.EAR` file), or an exploded archive directory representing the enterprise application (exploded `.EAR` directory).

Archive Versus Exploded Archive Directory

An exploded (unarchived) application directory lets you replace individual files without rebuilding the entire archive. Whether a particular change takes effect automatically or requires an update or redeployment depends on the resource type and configured reload behavior. For production update procedures, see Developing Applications for Production Redeployment. See

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

also Deployment Archive Files Versus Exploded Archive Directories in Deploying Applications to Oracle WebLogic Server. Developing Applications for Production Redeployment

You can also package applications in a single archived file, which is convenient for packaging modules and applications for distribution. Archive files are easier to copy, they use up fewer file handles than an exploded directory, and they can save disk space with file compression.

The Java classloader can search for Java class files (and other file types) in a JAR file the same way that it searches a directory in its classpath. Because the classloader can search a directory or a JAR file, you can deploy Jakarta EE modules on WebLogic Server in either a JAR (archived) file or an exploded (unarchived) directory.

wlpackage Ant Task Example

After compiling the split-directory application and running any additional precompilation or generation steps it requires, use wlpackage to produce either an EAR archive or a complete exploded EAR for distribution. The following illustrative build file continues the earA suite layout from Chapter 4; it is not the installed MedRec sample. Run it at the suite root after setting the WebLogic environment. source/earA and build/earA must already exist and contain the intended source/build outputs. Keep dist outside both input trees.

```
<project name="earA-packaging" default="package.ear" basedir=".">
  <taskdef name="wlpackage"
           classname="weblogic.ant.taskdefs.build.WLPackageTask"/>
  <property name="src.dir" location="source/earA"/>
  <property name="dest.dir" location="build/earA"/>
  <property name="dist.dir" location="dist"/>

  <target name="package.ear">
    <mkdir dir="${dist.dir}"/>
    <wlpackage srcdir="${src.dir}" destdir="${dest.dir}"
               tofile="${dist.dir}/earA.ear"/>
  </target>

  <target name="package.exploded">
    <delete dir="${dist.dir}/earA-exploded"/>
    <wlpackage srcdir="${src.dir}" destdir="${dest.dir}"
               todir="${dist.dir}/earA-exploded"/>
  </target>
</project>
```

Use ant -f package-build.xml package.ear or ant -f package-build.xml package.exploded after saving this example as package-build.xml. Choose the output format appropriate to your deployment procedure. package.exploded deletes only the previous generated dist/earA-exploded output before copying, so use that path exclusively for this package.

wlpackage Ant Task Attribute Reference

The following table describes the attributes of the `wlpackage` Ant task.

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

Table 6-1 Attributes of the wlpackage Ant Task

Attribute	Description	Data Type	Required?
`tofile`	Name of the EAR archive file into which the `wlpackage` Ant task packages the split development directory application.	File (path resolved relative to the Ant project base directory)	Specify exactly one of tofile or todir; setting both or neither is an error.
`todir`	Name of an exploded directory into which the `wlpackage` Ant task packages the split development directory application. Creates the output directory if needed. If the directory already exists, the task copies into it and does not delete files absent from the current source/build trees. For a complete release package, use a fresh generated output directory or remove the previous generated package directory before invoking wlpackage.	File (path resolved relative to the Ant project base directory)	Specify exactly one of tofile or todir; setting both or neither is an error.
`srcdir`	Specifies the source directory of your split development directory application. The source directory contains all editable files for your project—Java source files, editable descriptor files, JSPs, static content, and so forth.	File (path resolved relative to the Ant project base directory)	Yes.
`destdir`	Existing build directory containing the application's compiled classes and any other outputs required by the application. Run wlcompile first, and run wlappc or your custom generation steps where required. wlpackage combines the source/build trees; it does not compile Java sources or generate deployment descriptors.	File (path resolved relative to the Ant project base directory)	Yes.

For custom selections, use Ant fileset elements with dir, includes and excludes as appropriate; custom selections replace rather than extend the defaults described above.

Both srcdir and destdir must exist and be directories. Use an output path outside both input trees; see the packaging example.

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

7

Developing Applications for Production Redeployment

You can program and maintain applications with WebLogic Server using the production redeployment strategy.

What is Production Redeployment?

Production redeployment enables an administrator to redeploy a new version of an application in a production environment without stopping the deployed application or otherwise interrupting the application's availability to clients.

Production redeployment works by deploying a new version of an updated application alongside an older version of the same application. WebLogic Server automatically manages client connections so that only new client requests are directed to the new version. Clients already connected to the application during the redeployment continue to use the older, retiring version of the application until they complete their work.

See Using Production Redeployment to Upgrade Applications for more information.

Supported and Unsupported Application Types

Production redeployment supports HTTP and RMI clients, and the inbound JMS/message-driven-bean case described in Additional Application Support. Your development and design team must ensure that applications using production redeployment are not accessed by an unsupported client. Additional Application Support

See Additional Application Support.

WebLogic Server does not detect when unsupported clients access the application, and does not preserve unsupported client connections during production redeployment.

Enterprise applications can contain any of the supported Jakarta EE module types. Enterprise applications can also include application-scoped JMS and JDBC modules.

If an enterprise application includes a JCA resource adapter module, the module:

- Must implement the jakarta.resource.spi.ResourceAdapter lifecycle contract supported by this release.
- Both versions must implement weblogic.connector.extensions.Suspendable and report supportsVersioning() as true; the new version must also report supportsInit() as true.
- Both resource adapters must be modules of their EAR application, with enable-access-outside-app set to false.

WebLogic Server calls init(oldResourceAdapter, properties) on the new adapter before its start, then calls startVersioning(newResourceAdapter, properties) on the old adapter. The old adapter must notify the connector container through ExtendedBootstrapContext.complete() when it is safe to undeploy. Coordinate the adapter’s resource migration with these callbacks.

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

For a complete list of production redeployment requirements for resource adapters, see Production Redeployment in *Developing Resource Adapters for Oracle WebLogic Server*.

Additional Application Support

Additional production redeployment support is provided for enterprise applications that are accessed by inbound JMS messages from a global JMS destination, and that use one or more message-driven beans as consumers. For this type of application, WebLogic Server suspends message-driven beans in the older, retiring application version before deploying message-driven beans in the newer version. Production redeployment is not supported with JMS consumers that use the JMS API for global JMS destinations. If the message-driven beans need to receive all messages published from topics, including messages published while beans are suspended, use durable subscribers.

Programming Requirements and Conventions

WebLogic Server performs production redeployment by deploying two instances of an application simultaneously. You must observe certain programming conventions to ensure that multiple instances of the application can co-exist in a WebLogic Server domain.

The following sections describe each programming convention required for using production redeployment:

Applications Should Be Self-Contained

As a best practice, applications that use production redeployment should be self-contained in their use of resources. This means you should generally use application-scoped JMS and JDBC resources, rather than global resources, whenever possible for versioned applications.

If an application must use a global resource, you must ensure that the application supports safe, concurrent access by multiple instances of the application. This same restriction also applies if the application uses external (separately-deployed) applications, or uses an external property file. WebLogic Server does not prevent the use of global resources with versioned applications, but you must ensure that resources are accessed in a safe manner.

Looking up a global JNDI resource from within a versioned application results in a warning message. To disable this check, set the JNDI environment property `weblogic.jndi.WLContext.ALLOW_GLOBAL_RESOURCE_LOOKUP` to `true` when performing the JNDI lookup.

Similarly, looking up an external application results in a warning unless you set the JNDI environment property, `weblogic.jndi.WLContext.ALLOW_EXTERNAL_APP_LOOKUP`, to `true`.

WLContext exposes these Java constants for JNDI environment keys. In a Properties file or other string-key configuration, use the literal key. Set the selected property to the String value "true" only when the application meets the resource/version compatibility requirements; the default is false.

Table 7-1 JNDI Version-Lookup Environment Properties

Java constant	Literal environment key	Effect
WLContext.ALLOW_GLOBAL_RESOURCE_LOOKUP	weblogic.jndi.allowGlobalResourceLookup	Suppresses the versioning warning for the global JDBC/JMS resources checked by the naming service.

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

Table 7-1 (Cont.) JNDI Version-Lookup Environment Properties

Java constant	Literal environment key	Effect
WLContext.ALLOW_EXTERNAL_APP_LOOKUP	weblogic.jndi.allowExternalAppLookup	Suppresses the external-application versioning warning when a lookup has no version association for the target application.
WLContext.RELAX_VERSION_LOOKUP	weblogic.jndi.relaxVersionLookup	When the requested application version cannot be found, permits fallback to the active version.

Suppressing a warning does not coordinate shared-resource access or make different versions compatible.

Versioned Applications Access the Current Version JNDI Tree by Default

WebLogic Server binds application-scoped resources, such as JMS and JDBC application modules, into a local JNDI tree available to the application. As with non-versioned applications, versioned applications can look up application-scoped resources directly from this local tree. Application-scoped JMS modules can be accessed via any supported JMS interfaces, such as the JMS API or a message-driven bean.

Application modules that are bound to the global JNDI tree should be accessed only from within the same application version. WebLogic Server performs version-aware JNDI lookups and bindings for global resources deployed in a versioned application. By default, an internal JNDI lookup of a global resource returns bindings for the same version of the application.

If the current version of the application cannot be found, you can use the JNDI environment property `weblogic.jndi.WLContext.RELAX_VERSION_LOOKUP` to return bindings from the currently active version of the application, rather than the same version.

> **Note**
>
> Set `weblogic.jndi.WLContext.RELAX_VERSION_LOOKUP` to `true` only if you are certain that the newer and older version of the resource that you are looking up are compatible with one another.

Security Providers Must Be Compatible

Every authorization, role-mapping and credential-mapping provider configured in the application's security realm must support application versioning through the weblogic.security.spi.VersionableApplicationProvider SSPI. The default WebLogic Server security providers for authorization, role mapping, and credential mapping support the application versioning SSPI.

Applications Must Specify a Version Identifier

In order to use production redeployment, both the current, deployed version of the application and the updated version of the application must specify unique version identifiers. See Assigning an Application Version.

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

Applications Can Access Name and Identifier

Versioned applications can programmatically obtain both an application name, which remains constant across different versions, and an application identifier, which changes to provide a unique label for different versions of the application. Use the application name for basic display or error messages that refer to the application's name irrespective of the deployed version. Use the application ID when the application must provide a unique identifier for the deployed version of the application. See Accessing Version Information for more information about the MBean attributes that provide the name and identifier.

Client Applications Use Same Version when Possible

As described in What is Production Redeployment?, WebLogic Server attempts to route a client application's requests to the same version of the application until all of the client's in-progress work has completed. However, if an application version is retired using a timeout period, or is undeployed, the client's request will be routed to the active version of the application. In other words, a client's association with a given version of an application is maintained only on a "best-effort basis."

This behavior can be problematic for client applications that recursively access other applications when processing requests. WebLogic Server attempts to dispatch requests to the same versions of the recursively-accessed applications, but cannot guarantee that an intermediate application version is not undeployed manually or after a timeout period. If you have a group of related applications with strict version requirements, Oracle recommends packaging all of the applications together to ensure version consistency during production redeployment.

Assigning an Application Version

Specify a unique version identifier in the application's META-INF/MANIFEST.MF and update it for each application release. Production redeployment requires both the existing deployment and its replacement to be versioned, with different version identifiers, and to meet the supported-application and compatibility requirements. Use the production redeployment procedure and an appropriate retirement policy when introducing the new version.

For testing purposes, a deployer can also assign a version identifier to an application during deployment and redeployment. See Assigning a Version Identifier During Deployment and Redeployment in *Deploying Applications to Oracle WebLogic Server*.

Application Version Conventions

WebLogic Server obtains the application version from the value of the `Weblogic-Application-Version` property in the `MANIFEST.MF` file. The version string can be a maximum of 215 characters long, and must consist of valid characters as identified in Table 7-2.

The Deployer -appversion argument has a stricter limit in this release: it must contain fewer than 215 characters. Use at most 214 characters when the same version value must work through both paths.

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

Table 7-2 Application Version Character and Length Rules

Rule	Value
Allowed ASCII characters	A–Z, a–z, 0–9, period (.), underscore (_) and hyphen (-).
Disallowed complete version values	A single period (.) or two periods (..). These are whole-value restrictions, not a ban on periods inside other values.
Application version validator maximum	215 characters.
Deployer -appversion argument maximum	214 characters.

A production-redeployment version identifier must be nonempty. Underscore and hyphen are accepted individually, although meaningful release labels are easier to identify.

For example, the following manifest file content describes an application with version "`v920.beta`":

```
Manifest-Version: 1.0
Weblogic-Application-Version: v920.beta
```

Upgrading Applications to Use Production Redeployment

You can upgrade applications for deployment to WebLogic Server to use production redeployment.

An existing unversioned deployment cannot be converted into a versioned deployment by a side-by-side production redeployment. Establish the initial versioned deployment through a planned undeploy/deploy transition before using production redeployment for later versions; that initial transition does not provide the production-redeployment availability guarantee. For versioned deployments, AppDeploymentMBean.ApplicationName is the logical deployment name. AppDeploymentMBean.Name and ApplicationIdentifier identify the particular version. Use the attribute appropriate to the application's need; see Accessing Version Information.
Accessing Version Information

Accessing Version Information

Application code or management clients can use the following MBean attributes to read version information for display, logging or other uses.

AppDeploymentMBean provides the following read-only configuration attributes.

Table 7-3 Read-Only Version Attributes in AppDeploymentMBean

Attribute Name	Description
`ApplicationName`	Logical deployment name of the application, shared by its versions.
`VersionIdentifier`	Version identifier for this deployment; null for an unversioned application.
`ApplicationIdentifier`	Identifier for this particular application deployment/version; for an unversioned application it is the application name.

ApplicationRuntimeMBean provides the following read-only runtime attributes.

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

Table 7-4 Read-Only Version Attributes in ApplicationRuntimeMBean

Attribute Name	Description
`ApplicationName`	A String that represents the deployment name of the application
`ApplicationVersion`	A string that represents the version of the application.
`ActiveVersionState`	An int indicating this application version's activation role. Values defined by weblogic.deploy.version.AppActiveVersionState are: INACTIVE (0)—not the currently active version; it may not yet be activated or may be retiring. ACTIVE_ADMIN (1)—the active version for administrative-channel requests. ACTIVE (2)—the active version for normal, non-administrative-channel requests. These values are distinct from application/module lifecycle states.AppActiveVersionState constants

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

8

Using Jakarta EE Annotations and Dependency Injection

Learn about Jakarta EE metadata annotations and dependency injection (DI) in WebLogic Server.

Annotation Processing

Annotations simplify the application development process by allowing developers 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 (J2EE 1.4 and earlier).

Jakarta EE components use Java annotations to declare supported deployment metadata. Servlets execute in the web container; enterprise beans execute in the Jakarta Enterprise Beans container. Depending on the component and the configuration required, standard descriptors such as application.xml and web.xml can be omitted. Supply a descriptor when the application needs configuration that annotations and defaults do not provide. For current annotation packages and types, see the Jakarta EE 11 API documentation. Jakarta EE 11 API documentation

Annotation Parsing

The application components can use annotations to define their needs. Annotations reduce or eliminate the need to deal with deployment descriptors. Annotations simplify the development of application components. The deployment descriptor can still override values defined in the annotation. One usage of annotations is to define fields or methods that need Dependency Injection (DI). Annotations are defined on the POJO (plain old Java object) component classes like the EJB or the servlet.

For @Resource, a field or setter-method annotation declares an injection point. A class-level @Resource declares a named resource reference in the component environment without an injection target; obtain that reference through JNDI or a component-context lookup. On a class-level @Resource, specify both name and type explicitly. See Dependency Injection of Resources.

Deployment View of Annotation Configuration

WebLogic Server provides weblogic.deploy.api.tools.SessionHelper for deployment tools using the Java EE Deployment API (JSR 88). SessionHelper can expose the effective descriptor configuration, including configuration inferred from annotations, even when an input module omits a descriptor. Supported configuration changes can be saved in a deployment plan and applied during deployment. Only properties supported for plan-based configuration can be changed in this way. SessionHelper

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

Compiling Annotated Classes

The weblogic.appc and weblogic.appmerge command-line utilities process supported annotations in an application or module. Use -writeInferredDescriptors to write descriptors containing inferred deployment metadata. For web and EJB modules, the resulting standard module descriptors are marked metadata-complete="true". This suppresses the deployment-annotation processing governed by those descriptors; it does not disable every annotation facility, such as CDI. The wlappc Ant task supports annotation-aware compilation, but it does not expose a writeInferredDescriptors attribute.

When processing existing web/EJB module descriptors for inferred output, the tools write backup descriptors with the .orig suffix, for example WEB-INF/web.xml.orig or META-INF/ejb-jar.xml.orig. A descriptor absent from the input has no original copy to back up. Preserve an untouched input archive or source tree. To incorporate later annotation changes, run the tools against that original input; if reusing generated output, restore the applicable original descriptors first, and remove any generated descriptor that had no original input counterpart. Do not expect changed deployment annotations to override a previously generated metadata-complete descriptor.

For an enterprise application without application.xml, the tools can infer and write the application descriptor according to the Jakarta EE inference rules.

For more information on using `appc`, see weblogic.appc Reference. For more information on using `appmerge`, see Using weblogic.appmerge to Merge Libraries.

Dynamic Annotation Updates

A deployment-plan update changes supported deployment configuration. Use the update operation for plan changes; it is not a general replacement operation for application classes. Whether a configuration property can change dynamically depends on that property and its container. When application classes or their deployment annotations change, rebuild the application and use the applicable redeployment procedure. If using production redeployment, follow Developing Applications for Production Redeployment and its versioning and application-compatibility requirements. If the build uses inferred descriptors, first follow Compiling Annotated Classes so a metadata-complete descriptor does not mask the annotation changes. Developing Applications for Production Redeployment; Compiling Annotated Classes

Dependency Injection of Resources

Dependency injection (DI) allows application components to declare dependencies on external resources and configuration parameters via annotations. The container reads these annotations and injects resources or environment entries into the application components.

For supported container-managed components, resource injection can replace an explicit JNDI lookup in application code. Programmatic JNDI lookup remains available through the Java SE javax.naming APIs.

A field or a method of an application component can be annotated with the `@Resource` annotation. Note that the container will unbox the environment entry as required to match it to a primitive type used for the injection field or method. Example 8-1 illustrates how an application component uses the `@Resource` annotation to declare environment entries.

This stateless session bean receives two integer environment entries from the deployer. The numbers below are demonstration values, not tax rules.

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

Example 8-1 Dependency Injection of Environment Entries

```
package example;

import jakarta.annotation.Resource;
import jakarta.ejb.Stateless;

@Stateless
public class TaxPolicy {
    @Resource
    private int maxExemptions;

    @Resource
    private int minExemptions;

    public int getMaxExemptions() {
        return maxExemptions;
    }

    public int getMinExemptions() {
        return minExemptions;
    }
}
```

Supply these demonstration environment entries in META-INF/ejb-jar.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar 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/ejb-jar_4_0.xsd"
         version="4.0">
  <enterprise-beans>
    <session>
      <ejb-name>TaxPolicy</ejb-name>
      <env-entry>
        <env-entry-name>example.TaxPolicy/maxExemptions</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>10</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>example.TaxPolicy/minExemptions</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>0</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Because @Resource does not specify name, the container derives example.TaxPolicy/maxExemptions and example.TaxPolicy/minExemptions from the declaring class and field names. The descriptor supplies Integer values, which are unboxed into the int fields. Obtain the bean through the container; new TaxPolicy() does not request container injection. The field or method may have any access qualifier (public, private, etc.). For all classes except application client main classes, the fields or methods must not be static. The application client container invokes the client main class through its static main method rather than creating an instance of that main class. Injection fields and methods on that main class must therefore be static.

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

Application Life Cycle Annotation Methods

An application component may need to perform initialization of its own after all resources have been injected. To support this case, one method of the class can be annotated with the `@PostConstruct` annotation. This method will be called after all injections have occurred and before the class is put into service. This method will be called even if the class doesn't request any resources to be injected. Similarly, for classes whose life cycle is managed by the container, the `@PreDestroy` annotation can be applied to one method that will be called when the class is taken out of service and will no longer be used by the container. Each class in a class hierarchy may have `@PostConstruct` and `@PreDestroy` methods.

The order in which the methods are called matches the order of the class hierarchy, with methods on a superclass being called before methods on a subclass. For an application client main class, the container supports a static @PostConstruct callback before invoking main. It does not support @PreDestroy for the application client.

Standard Jakarta Annotations

The following sections describe the Jakarta Annotations APIs supported by the applicable Jakarta EE component containers.

For information about EJB-specific annotations for WebLogic Server EJBs, see *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

For information about Web component-specific annotations for WebLogic Server applications, see WebLogic Annotation for Web Components in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

jakarta.annotation.PostConstruct

Target: Method

Specifies the life cycle callback method that the application component should execute before the first business method invocation and after dependency injection is done to perform any initialization. This method will be called after all injections have occurred and before the class is put into service. This method will be called even if the class doesn't request any resources to be injected.

Declaring a @PostConstruct method is optional. Use it when the component needs initialization after injection. A class may declare at most one @PostConstruct method; different classes in the component's hierarchy can each declare a callback, subject to the applicable inheritance and overriding rules.

The method annotated with `@PostConstruct` must follow these requirements:

- A lifecycle callback declared on the target component class has signature void method() and must not throw checked exceptions. A lifecycle interceptor method declared in an interceptor class has signature void method(jakarta.interceptor.InvocationContext) or Object method(jakarta.interceptor.InvocationContext). A returned Object is ignored for a lifecycle event. A method that also intercepts business or timeout invocations may declare checked exceptions, but must not let a checked exception escape during a post-construct or pre-destroy invocation. See Jakarta Interceptors, Lifecycle Event Callbacks. Jakarta Interceptors 2.2, Lifecycle Event Callbacks
- The method may be `public`, `protected`, `package private` or `private`.
- The method must not be `static` except for the application client.

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

- Use a non-final callback method. A component specification may explicitly relax this restriction.
- If the method throws an unchecked exception, the class must not be put into service. In the case of EJBs, the method annotated with PostConstruct can handle exceptions and cleanup before the bean instance is discarded.

This annotation does not have any attributes.

jakarta.annotation.PreDestroy

Target: Method

Specifies the life cycle callback method that signals that the application component is about to be destroyed by the container. You typically apply this annotation to methods that release resources that the class has been holding.

Each class may declare at most one @PreDestroy callback. This callback is supported for container-managed objects whose full lifecycle the container controls; it is not supported for application clients.

The method annotated with `@PreDestroy` must follow these requirements:

- A lifecycle callback declared on the target component class has signature void method() and must not throw checked exceptions. A lifecycle interceptor method declared in an interceptor class has signature void method(jakarta.interceptor.InvocationContext) or Object method(jakarta.interceptor.InvocationContext). A returned Object is ignored for a lifecycle event. A method that also intercepts business or timeout invocations may declare checked exceptions, but must not let a checked exception escape during a post-construct or pre-destroy invocation. See Jakarta Interceptors, Lifecycle Event Callbacks. Jakarta Interceptors 2.2, Lifecycle Event Callbacks
- The method may be `public`, `protected`, `package private` or `private`.
- The method must not be static.
- Use a non-final callback method. A component specification may explicitly relax this restriction.
- An unchecked exception from a @PreDestroy callback is ignored under the Jakarta Annotations contract. For lifecycle interceptor chains, apply the Jakarta Interceptors exception rules; arrange cleanup so that it does not depend on a later callback executing successfully.

This annotation does not have any attributes.

jakarta.annotation.Resource

Target: Class, Method, Field

Specifies a dependence on an external resource, such as a JDBC data source or a JMS destination or connection factory.

If you specify the annotation on a field or method, the container injects an instance of the requested resource into the bean when the bean is initialized. If you apply the annotation to a class, the annotation declares a resource that the component will look up at runtime.

Attributes

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

Table 8-1 Attributes of the jakarta.annotation.Resource Annotation

Name	Description	Data Type	Required?
`name`	Names the logical resource reference in the component environment. Relative names are resolved under java:comp/env; lookup identifies the target resource separately. If you apply the `@Resource` annotation to a field, the default value of the `name` attribute is the field name, qualified by the fully qualified class name. If you apply it to a method, the default value is the component property name corresponding to the method, qualified by the fully qualified class name. If you apply the annotation to class, there is no default value and thus you are required to specify the attribute.	`String`	Yes on a class-level declaration; otherwise inferred when omitted.
lookup	JNDI name of an existing compatible resource to which this reference links. The default is an empty string (no explicit lookup target).	String	No
`type`	Specifies the Java data type of the resource. If you apply the `@Resource` annotation to a field, the default value of the `type` attribute is the type of the field. If you apply it to a method, the default is the type of the component property. If you apply it to a class, there is no default value and thus you are required to specify this attribute.	Class<?>	Yes on a class-level declaration; otherwise inferred when omitted.
`authenticationType`	Specifies the authentication type to use for the resource. Valid values for this attribute are: • `AuthenticationType.CONTAINER` • `AuthenticationType.APPLICATION` Default value is `AuthenticationType.CONTAINER` Applies to connection-factory resource references; do not explicitly specify it for other resource types.	`AuthenticationType`	No
`shareable`	Indicates whether a resource can be shared between this component and other components. Valid values for this attribute are `true` and `false`. Default value is `true`. Applies to connection-factory references; do not explicitly specify it for other resource types.	boolean	No

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

Table 8-1 (Cont.) Attributes of the jakarta.annotation.Resource Annotation

Name	Description	Data Type	Required?

`mappedName`	Specifies a WebLogic Server-specific name to which the component reference should be mapped.
	For resource references resolved through WebLogic Server's JNDI-name selection, an explicit JNDI-name mapping in the WebLogic descriptor takes precedence. Without that mapping, a lookup target is used when supplied; mappedName is the fallback when neither supplies a target. A mappedName is product-specific, so prefer lookup when an explicit JNDI reference to a configured compatible resource is appropriate.
	These fields belong inside a container-managed component. Import jakarta.annotation.Resource, java.net.URL, javax.sql.DataSource, jakarta.jms.ConnectionFactory and jakarta.jms.Queue. Configure the named JDBC and JMS resources before deployment. The URL is an illustrative value; this example does not open a network connection.
	`@Resource(mappedName = "https://example.com")`
	`private URL url;`
	`@Resource(mappedName = "customerDB")`
	`private DataSource db;`
	`@Resource(mappedName = "jms/ConnectionFactory")`
	`private ConnectionFactory connectionFactory;`
	`@Resource(mappedName = "jms/Queue")`
	`private Queue queue;`
	For resource references resolved through WebLogic Server's JNDI-name selection, an explicit JNDI-name mapping in the WebLogic descriptor takes precedence. Without that mapping, a lookup target is used when supplied; mappedName is the fallback when neither supplies a target. A mappedName is product-specific, so prefer lookup when an explicit JNDI reference to a configured compatible resource is appropriate.
		`String`	No

`description`	Specifies a description of the resource.	`String`	No

jakarta.annotation.Resources

Target: Class

@Resources groups multiple @Resource declarations on a class. @Resource is repeatable, so multiple class-level @Resource annotations are also supported; the compiler represents them through the @Resources container.

Attributes

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

Table 8-2 Attributes of the jakarta.annotation.Resources Annotation

Name	Description	Data Type	Required?
`value`	Specifies the array of `@Resource` annotations.	`Resource[]`	Yes

Standard Jakarta Security Annotations

The following sections describe Jakarta security annotations; support and behavior depend on the component technology.

Support is defined by the component specification. In the enterprise-bean case, @RolesAllowed, @PermitAll and @DenyAll control access to container-invoked business methods. They do not define servlet URL security constraints. For HTTP servlet access, use @ServletSecurity or web.xml security constraints as described in Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server. @DeclareRoles declares application roles, and @RunAs specifies an application execution identity where supported.

ServletSecurity

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

jakarta.annotation.security.DeclareRoles

Target: Class

Declares logical security roles used by this application component.

You typically use this annotation to define roles that can be tested from within the methods of the annotated class, such as using the `isUserInRole` method. You can also use the annotation to explicitly declare roles that are implicitly declared if you use the `@RolesAllowed` annotation on the class or a method of the class.

Declare application roles using the supported annotations or standard deployment descriptor. Configure how users and groups are assigned to those roles through the applicable WebLogic deployment mappings or configured role-mapping policy. Declaring a role does not create a user or group. For administrative role-mapping procedures, see Security Roles in the Oracle WebLogic Remote Console Online Help.

Attributes

Table 8-3 Attributes of the jakarta.annotation.security.DeclareRoles Annotation

Name	Description	Data Type	Required?
`value`	Array of logical application security role names to declare.	`String[]`	Yes

jakarta.annotation.security.DenyAll

Target: Class, Method

Excludes the affected container business methods from invocation. On a class, it establishes the default method-access rule for that class; a supported method-level access annotation takes precedence for that method.

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

This annotation does not have any attributes.

jakarta.annotation.security.PermitAll

Target: Class, Method

Marks the affected container business methods as unchecked for role-based authorization; callers need not belong to an application role. On a class, it establishes the default for that class; a supported method-level access annotation takes precedence for that method. This does not turn off independent authentication, transport or other security requirements.

This annotation does not have any attributes.

jakarta.annotation.security.RolesAllowed

Target: Class, Method

Specifies the list of security roles that are allowed to access methods in the Jakarta EE container.

If you specify it at the class-level, then it applies to all methods in the application component. If you specify it at the method-level, then it only applies to that method. If you specify the annotation at both the class- and method-level, the method value overrides the class value.

Declare application roles using the supported annotations or standard deployment descriptor. Configure how users and groups are assigned to those roles through the applicable WebLogic deployment mappings or configured role-mapping policy. Declaring a role does not create a user or group. For administrative role-mapping procedures, see Security Roles in the Oracle WebLogic Remote Console Online Help.

Attributes

Table 8-4 Attributes of the jakarta.annotation.security.RolesAllowed Annotation

Name	Description	Data Type	Required?
`value`	Array of logical application security role names permitted to invoke the affected methods.	`String[]`	Yes

jakarta.annotation.security.RunAs

Target: Class

@RunAs names the logical application security role used to determine a component's identity when it makes calls to other components. It does not change the identity of the caller being authorized to invoke the component. At deployment, map the run-as role to an appropriate user identity using the applicable WebLogic descriptor or configured role-mapping rules. For an explicit role-to-user mapping, use run-as-role-assignment with role-name and run-as-principal-name in weblogic.xml for a web module or weblogic-ejb-jar.xml for an EJB module. The run-as principal must identify a valid user, not just a group name. Role membership mappings may involve groups, but the effective run-as identity is a user.

Attributes

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

Table 8-5 Attributes of the jakarta.annotation.security.RunAs Annotation

Name	Description	Data Type	Required?
`value`	Logical application role name used to determine the component's run-as identity.	`String`	Yes

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

9

Using Contexts and Dependency Injection for the Jakarta EE Platform

WebLogic Server provides an implementation of the Contexts and Dependency Injection (CDI) specification. The CDI specification defines a set of services for using injection to specify dependencies in an application. CDI provides contextual life cycle management of beans, type-safe injection points, a loosely coupled event framework, loosely coupled interceptors and decorators, alternative implementations of beans, bean navigation through the Unified Expression Language (EL), and a service provider interface (SPI) that enables CDI extensions to support third-party frameworks or future Jakarta EE components.

About CDI for the Jakarta EE Platform

CDI for the Jakarta EE Platform specification was formerly called Web Beans. CDI injection simplifies the use of managed beans with JSF technology in Web applications.

WebLogic Server 26.1 implements Jakarta Contexts and Dependency Injection 4.1 for Jakarta EE 11. CDI integrates with Jakarta Dependency Injection 2.0, Jakarta Interceptors 2.2, Jakarta Annotations 3.0 and the applicable Jakarta EE component specifications. CDI managed beans are defined by CDI; they are distinct from the former Jakarta Managed Beans specification. Jakarta Contexts and Dependency Injection 4.1 Jakarta Dependency Injection 2.0 Jakarta Interceptors 2.2 Jakarta Annotations 3.0

CDI provides the following features:

- **Contexts.** This feature enables you to bind the life cycle and interactions of stateful components to well-defined but extensible life cycle contexts.
- **Dependency injection.** This feature enables you to inject components into an application in a type-safe way and to choose at deployment time which implementation of a particular interface to inject.

CDI is integrated with the major component technologies in Jakarta EE, namely:

- Servlets
- Jakarta Server Pages (JSP)
- Jakarta Server Faces (JSF)
- Jakarta Enterprise Beans (EJB)
- Jakarta EE Connector architecture (JCA)
- Web services

Such integration enables standard Jakarta EE objects, such as Servlets and EJB components, to use CDI injection for dependencies. CDI injection simplifies, for example, the use of managed beans with JSF technology in Web applications.

Some links in this chapter provide historical Java EE 8 tutorial background. For this release, use the jakarta.* APIs and CDI 4.1 discovery and configuration rules described here; consult the Jakarta CDI 4.1 specification for the current contract. Jakarta CDI 4.1 specification

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

See Introduction to Contexts and Dependency Injection for the Java EE Platform in the *Java EE 8 Tutorial*.

CDI 2.0 Examples

The optional WebLogic Server examples include demonstrations of asynchronous events, observer ordering and InterceptionFactory, features introduced in CDI 2.0. The sample directories retain their historical javaee8 names; the current source uses Jakarta APIs.

- Asynchronous Events – Demonstrates how to produce async events and how singleton EJBs can consume these events.
- Observer Ordering – Demonstrates how singleton EJBs can consume events according to the priority.
- Interception Factory – Demonstrates how to produce a class instance with adding the specified annotation dynamically by InterceptionFactory.

For more information, see the CDI 2.0 examples in the WebLogic Server distribution kit: `ORACLE_HOME\wlserver\samples\server\examples\src\examples\javaee8\cdi` where `ORACLE_HOME` represents the directory in which you installed WebLogic Server. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

CDI 1.1 Example

A sample originating with Java EE 7 demonstrates CDI features in the historically named javaee7 directory. Installed sample location: `ORACLE_HOME\wlserver\samples\server\examples\src\examples\javaee7\cdi` where `ORACLE_HOME` represents the directory in which you installed WebLogic Server. See Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*. These paths are present when the samples component is installed; see Sample Applications and Code Examples for installation and execution instructions.

Defining a Managed Bean

A managed bean is the basic component in a CDI application and defines the beans that CDI can create and manage.

A bean is a source of the objects that CDI can create and manage. See About Beans in *The Java EE 8 Tutorial* .

A CDI managed-bean class must satisfy the CDI 4.1 managed-bean requirements, including the constructor, nesting and class-kind restrictions. It may have a no-argument constructor or an appropriate @Inject constructor. See Which Java Classes Are Managed Beans? in the CDI 4.1 specification. Which Java Classes Are Managed Beans?

> **Note**
>
> Eligibility and discovery are separate. With annotated discovery, give the class a bean-defining annotation, such as @Dependent or a normal scope. @Named, @Alternative and a custom qualifier by themselves are not bean-defining annotations. An explicit bean archive with bean-discovery-mode=all can discover eligible classes without such annotations.

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

Injecting a Bean

To use the beans that you define, inject them into another bean that an application such as a Jakarta Server Faces can use.

See Injecting Beans in *The Java EE 8 Tutorial* .

CDI ensures type-safe injection of beans by selecting the bean class on the basis of the Java type that is specified in the injection point, not the bean name. CDI also determines where to inject a bean from the Java type in the injection point.

In this respect, CDI bean injection is different than the resource injection that was introduced in the Java EE 5 specification, which selects the resource to inject from the string name of the resource. For example, a data source that is injected with the jakarta.annotation.Resource annotation is identified by its string name.

Create an injection point by annotating an instance field, a bean constructor, or an initializer method with jakarta.inject.Inject. Constructor and initializer-method parameters receive their dependencies because their enclosing constructor or method is annotated; annotate the parameters with qualifiers when selection requires them.

Example 9-1 shows how to use the `@Inject` annotation to inject a bean into another bean.

Example 9-1 Injecting a Bean into Another Bean

This example annotates an instance class field to inject an instance of the bean class `Greeting` into the class `Printer`.

```
import jakarta.inject.Inject;
...
public class Printer {
    @Inject Greeting greeting;
    ...
}
```

Defining the Scope of a Bean

A scope determines the lifecycle of a bean's contextual instances and how references resolve to those instances. Choose it for the state and lifetime required by the application; using injection does not itself require conversational state.

To define the scope of a bean, annotate the class declaration of the bean with the scope. The `jakarta.enterprise.context` package defines the following scopes:

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`
- `@Dependent`

For information about these scopes, see Using Scopes in *The Java EE 8 Tutorial* .

If no explicit scope or stereotype-supplied default scope applies, the bean's default scope is @Dependent. A dependent instance injected into another contextual instance is normally destroyed with its owning instance. The other listed scopes are normal scopes, generally

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

accessed through client proxies that resolve the instance for the active context. A session-scoped instance, for example, belongs to an HTTP session.

For a normal-scoped bean, obtaining an injected reference need not immediately create its contextual instance; the proxy obtains the instance when used in the active context.

Example 9-2 shows how to define the scope of a bean.

Example 9-2 Defining the Scope of a Bean

This example defines the scope of the `Accountant` bean class to be `@RequestScoped`.

The `Accountant` class in this example is qualified by the `@BeanCounter` qualifier. For more information, see Using Qualifiers.

```
package com.example.managers;

import jakarta.enterprise.context.RequestScoped;

@RequestScoped
@BeanCounter
public class Accountant implements Manager
{
 ...
}
```

Overriding the Scope of a Bean at the Point of Injection

Overriding the scope of a bean at the point of injection enables an application to request a new instance of the bean with the default scope `@Dependent`. The `@Dependent` scope specifies that the bean's life cycle is the life cycle of the object into which the bean is injected.

The CDI container provides no other life cycle management for the instance. For more information about scopes, see Defining the Scope of a Bean.

> **Note**
>
> The effects of overriding the scope of a bean may be unpredictable and undesirable, particularly if the overridden scope is `@Request` or `@Session`.

To override the scope of a bean at the point of injection, inject the bean by using the `javax.enterprise.inject.New` annotation instead of the `@Inject` annotation. For more information about the `@Inject` annotation, see Injecting a Bean.

Using Qualifiers

Qualifiers enable you to provide more than one implementation of a particular bean type.

When you use qualifiers, you select between implementations at development time. See Using Qualifiers in *The Java EE 8 Tutorial* .

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

> **Note**
>
> To select between alternative implementations at deployment time, use alternatives as explained in Providing Alternative Implementations of a Bean Type.

Using qualifiers involves the tasks that are explained in the following sections:

Defining Qualifiers for Implementations of a Bean Type

A qualifier is an application-defined annotation that enables you to identify an implementation of a bean type. Define a qualifier for each implementation of a bean type that you are providing.

Define qualifiers when the application needs to distinguish beans or injection requirements. Qualifiers can be used with alternatives and may be shared by several beans. Every bean has @Any. A bean also has @Default when it declares no qualifier other than @Named and/or @Any.

> **Note**
>
> CDI does not require a qualifier to be unique to a particular bean. You can define a qualifier to use for more than one bean type.

To define a qualifier:

1. Define a Java annotation type to represent the qualifier.
2. Annotate the declaration of the annotation type with the `jakarta.inject.Qualifier` annotation.
3. Specify that the qualifier is to be retained by the virtual machine at run time.

   Use the `java.lang.annotation.Retention(RUNTIME)` meta-annotation for this purpose.
4. Specify that the qualifier may be applied to the program elements `METHOD`, `FIELD`, `PARAMETER`, and `TYPE`.

   Use the `java.lang.annotation.Target({METHOD, FIELD, PARAMETER, TYPE})` meta-annotation for this purpose.

The following examples show how to define qualifiers `@BeanCounter` and `@PeopleManager` for different implementations of the same bean type.

Example 9-3 Defining the @BeanCounter Qualifier

This example defines the `@BeanCounter` qualifier.

```
package com.example.managers;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;
```


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

```
import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface BeanCounter {}
```

Example 9-4 Defining the @PeopleManager Qualifier

This example defines the `@PeopleManager` qualifier.

```
package com.example.managers;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PeopleManager {}
```

Applying Qualifiers to a Bean

Apply qualifiers to a bean class or producer to declare its selection metadata. An injection point without an explicit qualifier requires @Default; its Inject annotation is not a qualifier.

> **Note**
>
> CDI does not require a qualifier to be unique to a particular bean. You can apply the same qualifier to different types of beans in the set of beans that are available in the application.

To apply qualifiers to a bean, annotate the class declaration of the bean with each qualifier to apply. Any qualifier that you apply to a bean must be defined as explained in Defining Qualifiers for Implementations of a Bean Type.

The following examples show how to apply the qualifiers `@BeanCounter` and `@PeopleManager` to different implementations of the `Manager` bean type.

Example 9-5 Applying the @BeanCounter Qualifier to a Bean

This example applies the `@BeanCounter` qualifier to the `Accountant` class. The `Accountant` class is an implementation of the `Manager` bean type. The `@BeanCounter` qualifier is defined in Example 9-3.

```
package com.example.managers;
...
@BeanCounter
```


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

```
public class Accountant implements Manager
{...}
```

Example 9-6 Applying the@ PeopleManager Qualifier to a Bean

This example applies the `@PeopleManager` qualifier to the `Boss` class. The `Boss` class is an implementation of the `Manager` bean type. The `@PeopleManager` qualifier is defined in Example 9-4.

```
package com.example.managers;
...
@PeopleManager
public class Boss implements Manager
{...}
```

Injecting a Qualified Bean

To inject a qualified bean, create an injection point and annotate the injection point with the bean's qualifiers. The qualifiers at the injection point define the overall requirements of the injection target. The CDI application must contain a CDI managed bean that matches the type of the injection point and the qualifiers with which the injection point is annotated. Otherwise, a deployment error occurs. For more information about how to create an injection point, see Injecting a Bean.

If you do not annotate the injection point, the predefined qualifier `@Default` is applied to the injection point by default.

CDI resolves the injection point by first matching the bean type and then matching implementations of that type with the qualifiers in the injection point.

For each injection point, CDI finds available enabled beans assignable to the required type and matching all required qualifiers, then applies the alternative/priority ambiguity-resolution rules. An unsatisfied dependency or an ambiguity that those rules cannot resolve is a deployment problem. Selecting an alternative does not globally disable every other bean of that Java type; another qualifier can still select a different bean.

For information about alternatives, see Providing Alternative Implementations of a Bean Type.

Example 9-7 shows how to inject a qualified bean.

Example 9-7 Injecting a Qualified Bean

This example injects the `@BeanCounter` implementation of the `Manager` bean type. The `Manager` bean type is implemented by the following classes:

- `Accountant`, which is shown in Example 9-5
- `Boss`, which is shown in Example 9-6

In this example, the `Accountant` class is injected because the bean type and qualifier of this class match the bean type and qualifier in the injection point.

```
package com.example.managers;
...
import jakarta.inject.Inject;
...
public class PennyPincher {
   @Inject @BeanCounter Manager accountant;
   ...
}
```


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

Providing Alternative Implementations of a Bean Type

The environments for the development, testing, and production deployment of an enterprise application may be very different. Differences in configuration, resource availability, and performance requirements may cause bean classes that are appropriate to one environment to be unsuitable in another environment. By providing alternative implementations of a bean type, you can modify an application at deployment time to meet such differing requirements.

Different deployment scenarios may also require different business logic in the same application. For example, country-specific sales tax laws may require country-specific sales tax business logic in an order-processing application.

CDI enables you to select from any number of alternative bean type implementations for injection instead of a corresponding primary implementation. See Using Alternatives in CDI Applications in *The Java EE 8 Tutorial* .

> **Note**
>
> To select between alternative implementations at development time, use qualifiers as explained in Using Qualifiers.

Providing alternative implementations of a bean type involves the tasks that are explained in the following sections:

Defining an Alternative Implementation of a Bean Type

To define an alternative implementation of a bean type:

1. Write a bean class of the same bean type as primary implementation of the bean type.

   To ensure that any alternative can be injected into an application, you must ensure that all alternatives and the primary implementation are all of the same bean type. For information about how to inject a bean, see Injecting a Bean.

2. Annotate the class declaration of the implementation with the `jakarta.enterprise.inject.Alternative` annotation.

   > **Note**
   >
   > To ensure that the primary implementation is selected by default, do not annotate the class declaration of the primary implementation with `@Alternative`.

The following examples show the declaration of the primary implementation and an alternative implementation of a bean type. The alternative implementation is a mock implementation that is intended for use in testing.

Example 9-8 Declaring a Primary Implementation of a Bean Type

This example declares the primary implementation `OrderImpl` of the bean type `Order`.

```
package com.example.orderprocessor;
...
public class OrderImpl implements Order {
```


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

```
 ...
}
```

Example 9-9 Declaring an Alternative Implementation of a Bean Type

This example declares the alternative implementation `MockOrderImpl` of the bean type `Order`. The declaration of the primary implementation of this bean type is shown in Example 9-8.

```
package com.example.orderprocessor;
...
import jakarta.enterprise.inject.Alternative;

@Alternative
public class MockOrderImpl implements Order {
 ...
}
```

Selecting an Alternative Implementation of a Bean Type for an Application

An @Alternative bean is disabled unless selected. Listing an alternative class in a bean archive's beans.xml selects it for that bean archive. For application-wide selection, an alternative can instead use jakarta.annotation.Priority according to the CDI rules. The alternative must match the required types and qualifiers.

To select an alternative for a bean archive:

1. Add a `class` element for the alternative to the `alternatives` element in the `beans.xml` file.
2. In the `class` element, provide the fully qualified class name of the alternative.

For more information about the `beans.xml` file, see Configuring a CDI Application.

Example 9-10 shows a `class` element in the `beans.xml` file for selecting an alternative implementation of a bean type.

Example 9-10 Selecting an Alternative Implementation of a Bean Type

This example selects the alternative implementation `com.example.orderprocessor.MockOrderImpl`.

```
...
<alternatives>
    <class>com.example.orderprocessor.MockOrderImpl</class>
</alternatives>
...
```

This descriptor-based example intentionally demonstrates archive-local selection; do not also enable it through priority merely to repeat this example.

Applying a Scope and Qualifiers to a Session Bean

CDI enables you to apply a scope and qualifiers to a session bean.

A session bean is an EJB component that meets either of the following requirements:

- The class that implements the bean is annotated with one of the following annotations:
  - `jakarta.ejb.Singleton`, which denotes a singleton session bean
  - `jakarta.ejb.Stateful`, which denotes a stateful session bean
  - `jakarta.ejb.Stateless`, which denotes a stateless session bean

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

- The bean is listed in the `ejb-jar.xml` deployment-descriptor file.

For more information about session beans, see:

Developing Enterprise JavaBeans Using Deployment Descriptors for Oracle WebLogic Server

See the following sections:

Applying a Scope to a Session Bean

The scopes that CDI allows you to apply to a session bean depend on the type of the session bean as shown in Table 9-1.

Table 9-1 Allowed CDI Scopes for Session Beans

Session Bean Type	Allowed Scopes
Singleton	Either of the following scopes: • Dependent • Application
Stateful	Any
Stateless	Dependent

For more information about scopes in CDI, see Defining the Scope of a Bean.

When CDI injects a reference to a stateful session bean, CDI creates the bean, injects the bean's fields, and manages the stateful session bean according to its scope. When the context is destroyed, CDI calls the stateful session bean's remove method to remove the bean.

Applying Qualifiers to a Session Bean

CDI allows you to apply any qualifier to a session bean. CDI does not restrict the type of qualifier that you can apply to a session bean. For more information about qualifiers in CDI, see Using Qualifiers.

Using Producer Methods, Disposer Methods, and Producer Fields

A producer method is a method that generates an object that can then be injected. A disposer method enables an application to perform customized cleanup of an object that a producer method returns. A producer field is a field of a bean that generates an object.

A producer field is a simpler alternative to a producer method.

See Jakarta Contexts and Dependency Injection: Advanced Topics in the *Jakarta EE Tutorial*.

Defining a Producer Method

A producer method enables an application to customize how CDI managed beans are created. This customization involves overriding the process that CDI normally uses to resolve beans. A producer method enables you to inject an object that is not an instance of a CDI bean class.

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

A producer method must be a method of a CDI bean class or session bean class. However, a producer method may return objects that are not instances of CDI bean classes. In this situation, the producer method must return an object that matches a bean type.

A producer method can have any number of parameters. If necessary, you can apply qualifiers to these parameters. All parameters of a producer method are injection points. Therefore, the parameters of a producer method do not require the `@Inject` annotation.

To define a producer method, annotate the declaration of the method with the `jakarta.enterprise.inject.Produces` annotation.

If the producer method sometimes returns null, set the scope of the method to dependent.

> **Note**
>
> Calling a producer method directly in application code does not invoke CDI.

For an example of the definition of a producer method, see Example 9-11.

Defining a Disposer Method

A disposer method performs cleanup for objects produced by producer methods or producer fields declared by the same bean class. Mark exactly one disposed parameter with jakarta.enterprise.inject.Disposes. CDI resolves a disposer using the disposed parameter's required type and qualifiers and the CDI assignability rules. A disposer can match several producers in that class; a producer must not resolve to more than one disposer. Other disposer parameters are injection points.

Example 9-11 shows how to use the `@Produces` annotation to define a producer method and the `@Disposes` annotation to define a disposer method.

Configure a JDBC data source bound as jdbc/CustomerDB before deploying this example. CDI discovers SomeClass through @Dependent. On the first use requiring a contextual Connection during a request, connect obtains a connection from that configured data source. The connection is request-scoped; its matching disposer closes it when that contextual instance is destroyed. The data source itself remains container-managed. This example does not define application authentication or transaction boundaries.

Example 9-11 Defining a Producer Method and Disposer Method

```
package com.example.resources;
import jakarta.annotation.Resource;
import jakarta.enterprise.context.Dependent;
import jakarta.enterprise.context.RequestScoped;
import jakarta.enterprise.inject.Disposes;
import jakarta.enterprise.inject.Produces;
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;

@Dependent
public class SomeClass {
    @Resource(lookup = "jdbc/CustomerDB")
    private DataSource dataSource;

    @Produces @RequestScoped
    public Connection connect() throws SQLException {
```


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

```
        return dataSource.getConnection();
    }

    public void close(@Disposes Connection connection) throws SQLException {
        connection.close();
    }
}
```

Defining a Producer Field

A producer field is a simpler alternative to a producer method. A producer field must be a field of a managed bean class or session bean class. A producer field may be either static or nonstatic, subject to the following constraints:

- In a session bean class, the producer field must be a static field.
- In a managed bean class, the producer field can be either static or nonstatic.

To define a producer field, annotate the declaration of the field with the `jakarta.enterprise.inject.Produces` annotation.

If the producer field may contain a null when accessed, set the scope of the field to dependent.

> **Note**
>
> Using a producer field directly in application code does not invoke CDI.

Producer fields can have matching disposer methods under the same resolution rules.

Initializing and Preparing for the Destruction of a Managed Bean

CDI managed bean classes and their superclasses support the annotations for initializing and preparing for the destruction of a managed bean.

These annotations are defined in Jakarta Annotations 3.0: Common Annotations for the Java Platform. For more information, see Using Jakarta EE Annotations and Dependency Injection.

Initializing a Managed Bean

Initializing a managed bean specifies the life cycle callback method that the CDI framework should call after dependency injection but before the class is put into service.

To initialize a managed bean:

1. In the managed bean class or any of its superclasses, define a method that performs the initialization that you require.
2. Annotate the declaration of the method with the `jakarta.annotation.PostConstruct` annotation.

   When the managed bean is injected into a component, CDI calls the method after all injection has occurred and after all initializers have been called.

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

> **Note**
>
> As mandated by JSR 250, if the annotated method is declared in a superclass, the method is called unless a subclass of the declaring class overrides the method.

Preparing for the Destruction of a Managed Bean

Preparing for the destruction of a managed bean specifies the life cycle callback method that signals that an application component is about to be destroyed by the container.

To prepare for the destruction of a managed bean:

1. In the managed bean class or any of its superclasses, define a method that prepares for the destruction of the managed bean.

   In this method, perform any cleanup that is required before the bean is destroyed, such a releasing resources that the bean has been holding.

2. Annotate the declaration of the method with the `jakarta.annotation.PreDestroy` annotation.

   CDI calls the method before starting the logic for destroying the bean.

> **Note**
>
> As mandated by JSR 250, if the annotated method is declared in a superclass, the method is called unless a subclass of the declaring class overrides the method.

Intercepting Method Invocations and Life Cycle Events of Bean Classes

Intercepting a method invocation or a life cycle event of a bean class interposes an interceptor class in the invocation or event. When an interceptor class is interposed, additional actions that are defined in the interceptor class are performed.

An interceptor class simplifies the maintenance of code for tasks that are frequently performed and are separate from the business logic of the application. Examples of such tasks are logging and auditing.

> **Note**
>
> The programming model for interceptor classes is optimized for operations that are separate from the business logic of the application. To intercept methods that perform operations with business semantics, use a decorator class as explained in Decorating a Managed Bean Class.

The interceptors that were introduced in the Java EE 5 specification are specific to EJB components. For more information about Java EE 5 interceptors, see Specifying Interceptors for Business Methods or Life Cycle Callback Events in *Developing Enterprise JavaBeans for Oracle WebLogic Server*.

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

CDI enables you to use interceptors with the following types of Jakarta EE managed objects:

- CDI managed beans
- EJB session beans
- EJB message-driven beans

> **Note**
>
> You **cannot** use interceptors with EJB entity beans because CDI does not support EJB entity beans.

See Using Jakarta EE Interceptors in the *Jakarta EE Tutorial*.

Intercepting method invocations and life cycle events of bean classes involves the tasks that are explained in the following sections:

Defining an Interceptor Binding Type

An interceptor binding type is an application-defined annotation that associates an interceptor class with an intercepted bean. Define an interceptor binding type for each type of interceptor that you require.

com.example.billpayment.interceptor.Transactional is an application-defined binding used only to illustrate binding resolution. It is different from jakarta.transaction.Transactional and does not itself supply transaction semantics.

> **Note**
>
> CDI does not require an interceptor binding type to be unique to a particular interceptor class. You can define an interceptor binding type to use for more than one interceptor class.

To define an interceptor binding type:

1. Define a Java annotation type to represent the interceptor binding type.
2. Annotate the declaration of the annotation type with the `jakarta.interceptor.InterceptorBinding` annotation.
3. Specify that the interceptor binding type is to be retained by the virtual machine at run time.

   Use the `java.lang.annotation.Retention(RUNTIME)` meta-annotation for this purpose.
4. Specify that the interceptor binding type may be applied to the program elements `METHOD` and `TYPE`.

   Use the `java.lang.annotation.Target({METHOD, TYPE})` meta-annotation for this purpose.

Example 9-12 Defining An Interceptor Binding Type

This example defines the `@Transactional` interceptor binding type.

```
package com.example.billpayment.interceptor;
```


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

```
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import jakarta.interceptor.InterceptorBinding;

@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {}
```

Defining an Interceptor Class

An interceptor class is used to interpose in method invocations or life cycle events that occur in an associated target bean class. In an interceptor class, provide the code for tasks that are frequently performed and are separate from the business logic of the application, such as logging and auditing.

To define an interceptor class:

1. Define a Java class to represent the interceptor.
2. Annotate the declaration of the class with the following annotations:
   - `jakarta.interceptor.Interceptor`
   - The interceptor binding types that are defined for the class

     You can apply any number of interceptor binding types to an interceptor class.

     > **Note**
     >
     > CDI does not require an interceptor binding type to be unique to a particular interceptor class. You can apply the same interceptor binding type to multiple interceptor classes.
3. Implement the interceptor methods in the class.

   CDI does not require the signature of an interceptor method to match the signature of the intercepted method.
4. Identify the interceptor methods in the class.

   An interceptor method is the method that is invoked when a method invocation or a life cycle event of a bean class is intercepted.

   To identify an interceptor method, annotate the declaration of the method with the appropriate annotation for the type of the interceptor method.

Interceptor Method Type	Annotation
Method invocation	`jakarta.interceptor.AroundInvoke`
EJB timeout	`jakarta.interceptor.AroundTimeout`
Initialization of a managed bean or EJB component	`jakarta.annotation.PostConstruct`

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

Interceptor Method Type	Annotation
Destruction of a managed bean or EJB component	`jakarta.annotation.PreDestroy`
Activation of a stateful session bean	`jakarta.ejb.PostActivate`
Passivation of a stateful session bean	`jakarta.ejb.PrePassivate`

> **Note**
>
> An interceptor class can have multiple interceptor methods. However, an interceptor class can have no more than one interceptor method of a given type.

Example 9-13 shows how to define an interceptor class.

This pass-through example continues the invocation and returns its result. Use the applicable Jakarta Transactions/EJB facilities when actual transaction boundaries are required.

Example 9-13 Defining an Interceptor Class

This example defines the interceptor class for which the `@Transactional` interceptor binding type is defined. The `manageTransaction` method of this class is an interceptor method. The `@Transactional` interceptor binding is defined in Example 9-12.

```
package com.example.billpayment.interceptor;
import jakarta.interceptor.AroundInvoke;
import jakarta.interceptor.Interceptor;
import jakarta.interceptor.InvocationContext;

@Transactional
@Interceptor
public class TransactionInterceptor {
    @AroundInvoke
    public Object manageTransaction(InvocationContext context) throws Exception {
        return context.proceed();
    }
}
```

Identifying Methods for Interception

Identifying methods for interception associates the methods with the interceptor that is invoked when the methods are invoked. CDI enables you to identify all methods of a bean class or only individual methods of a bean class for interception.

- To identify all methods of a bean class for interception, annotate the declaration of the bean class with the appropriate interceptor binding type.
- To identify an individual method of a bean class for interception, annotate the declaration of the method with the appropriate interceptor binding type.

CDI does not require the signature of an intercepted method to match the signature of the interceptor method. To determine the arguments and return type of an intercepted method, an interceptor must query an interceptor context. Interception applies to eligible container invocations and supported lifecycle callbacks under the CDI and Interceptors rules; it does not intercept arbitrary direct Java calls or every private/static/final method.

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

> **Note**
>
> For interceptor-binding resolution, place the binding on the interceptor class and on the target class or eligible target method. The interceptor method is identified by @AroundInvoke or the applicable lifecycle annotation; it does not need the binding annotation. Alternatively, @Interceptors names interceptor classes directly on the supported target class or method. Bindings provide the indirect association illustrated here.

Example 9-14 Identifying All Methods of a Bean Class for Interception

This example identifies all methods of the `ShoppingCart` class for interception by the `@Transactional` interceptor.

```
package com.example.billpayment.interceptor;

@Transactional
public class ShoppingCart {
    ...
}
```

Example 9-15 Identifying an Individual Method of a Class for Interception

This example identifies only the `checkout` method of the `ShoppingCart` class for interception by the `@Transactional` interceptor.

```
package com.example.billpayment.interceptor;

public class ShoppingCart {
    ...
    @Transactional public void checkout() {
    ...
    }
}
```

Enabling an Interceptor

By default, an interceptor is disabled. If you require an interceptor to be interposed in method invocations and events, you must enable the interceptor explicitly.

The following procedure enables this class for a bean archive using beans.xml. Alternatively, @Priority enables it for the application and determines its priority ordering under the CDI rules.

To enable an interceptor:

1. Add a `class` element for the interceptor to the `interceptors` element in the `beans.xml` file.
2. In the `class` element, provide the fully qualified class name of the interceptor.

   For classes enabled only through this beans.xml list, ensure that the order of t he class elements in the beans.xml file matches the order in which the interceptors are to be invoked.

   For classes enabled only through this beans.xml list, CDI interceptors are invoked in the order in which they are declared in the beans.xml file. Interceptors that are defined in the ejb-jar.xml file or by the jakarta.interceptor.Interceptors annotation are called before the CDI interceptors. Interceptors are called before CDI decorators.

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

Priority-enabled interceptors are ordered by priority and are considered before interceptors enabled only through the bean archive's beans.xml list. Interceptors are invoked before decorators.

> **Note**
>
> Java EE 5 interceptors are invoked in the order in which they are annotated on an intercepted method.

For more information about the `beans.xml` file, see Configuring a CDI Application.

Example 9-16 shows a `class` element in the `beans.xml` file for enabling an interceptor class.

Example 9-16 Enabling an Interceptor Class

This example enables the interceptor class `com.example.billpayment.interceptor.TransactionInterceptor`. The interceptor class is defined in Example 9-13.

```
...
<interceptors>
    <class>com.example.billpayment.interceptor.TransactionInterceptor</class>
</interceptors>
...
```

Applying an Interceptor on a Producer

In CDI 1.x, the interceptor is not bound to a producer bean. CDI 2.0 introduces the interface `jakarta.enterprise.inject.spi.InterceptionFactory<T>`, which allows to apply interceptors programmatically to the return value of a producer method.

The `InterceptionFactory` interface allows to create a wrapper instance whose method invocations are intercepted by method interceptors and forwarded to a provided instance.

API outline; package jakarta.enterprise.inject.spi

```
public interface InterceptionFactory<T> {
    InterceptionFactory<T> ignoreFinalMethods();
    AnnotatedTypeConfigurator<T> configure();
    T createInterceptedInstance(T instance);
}
```

A producer method can receive InterceptionFactory<T> as an injected parameter, as below. Programmatic callers use BeanManager.createInterceptionFactory(CreationalContext<T>, Class<T>). An injected InterceptionFactory parameter is supported on a producer method, not an arbitrary bean field.

Producer fragment: Product and ActionBinding.Literal are application-defined collaborators. Supply their definitions and interceptor enablement before using this fragment; it is not a standalone compilation unit.

```
@Produces
@RequestScoped
public Product createInterceptedProduct(InterceptionFactory<Product>
interceptionFactory) {
```


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

```
  interceptionFactory.configure().add(ActionBinding.Literal.INSTANCE);
  return interceptionFactory.createInterceptedInstance(new Product());
}
```

See Using Interceptors in CDI Applications in *Java EE 8 Tutorial* for more information about using interceptors.

Decorating a Managed Bean Class

Decorating a managed bean class enables you to intercept invocations of methods in the decorated class that perform operations with business semantics.

A decorator decorates eligible bean types exposed through its decorated interfaces. The bean must match the delegate injection point's required type and qualifiers and satisfy the CDI decoration restrictions.

> **Note**
>
> The programming model for decorator classes is optimized for operations that perform the business logic of the application. To intercept methods that are separate from the business logic of an application, use an interceptor class as explained in Intercepting Method Invocations and Life Cycle Events of Bean Classes.

See Using Decorators in *The Java EE 8 Tutorial* .

Decorating a managed bean class involves the tasks that are explained in the following sections:

Defining a Decorator Class

A decorator class intercepts invocations of methods in the decorated class that perform operations with business semantics. A decorator class and an interceptor class are similar because both classes provide an around-method interception. However, a method in a decorator class has the same signature as the intercepted method in the decorated bean class.

To define a decorator class:

1. Write a Java class that implements the same interface as the bean class that you are decorating.

   If you want to intercept only some methods of the decorated class, declare the decorator class as an abstract class. If you declare the class as abstract, you are not required to implement all the methods of the bean class that you are decorating.

2. Annotate the class declaration of the decorator class with the `jakarta.decorator.Decorator` annotation.

3. Implement the methods of the decorated bean class that you want to intercept.

   If the decorator class is a concrete class, you must implement all the methods of the bean class that you are decorating.

   You must ensure that the intercepting method in a decorator class has the same signature as the intercepted method in the decorated bean class.

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

4. Add a delegate injection point to the decorator class.

   A decorator class must contain exactly one delegate injection point. A delegate injection point injects a delegate object, which represents the next element in the decorator chain or the underlying bean, into the decorator object.

   You can customize how any method in the decorator object handles the implementation of the decorated method. CDI allows but does not require the decorator object to invoke the corresponding delegate object. Therefore, you are free to choose whether the decorator object invokes the corresponding delegate object.

   a. Declare an injection point of the decorated interface type.

   b. Annotate the injection point with the `jakarta.decorator.Delegate` annotation.

   c. Apply qualifiers that you require to the injection point, if any.

      If you apply qualifiers to the injection point, the decorator applies only to beans whose bean class matches the qualifiers of the injection point.

> **Note**
>
> No special declaration, such as an annotation, is required to define a decorated bean class. An enabled decorator class applies to any bean class or session bean that matches the bean type and qualifiers of the delegate injection point.

Example 9-17 shows the definition of a decorator class.

Example 9-17 Defining a Decorator Class

This example defines the decorator class `DataAccessAuthDecorator`. This class decorates any bean of type `DataAccess`.

Because only some methods of the decorated class are to be intercepted, the class is declared as an abstract class. This class injects a delegate instance `delegate` of the decorated implementation of the `DataAcess` bean type.

```
import jakarta.decorator.*;
import jakarta.inject.Inject;
import java.lang.Override;
...
@Decorator
public abstract class DataAccessAuthDecorator
        implements DataAccess {

    @Inject @Delegate DataAccess delegate;

    @Override
    public void delete(Object object) {
        authorize(SecureAction.DELETE, object);
        delegate.delete(object);
    }

    private void authorize(SecureAction action, Object object) {
        ...
    }
}
```


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

Enabling a Decorator Class

By default, a decorator class is disabled. If you require a decorator class to be invoked in a CDI application, you must enable the decorator class explicitly.

The following procedure enables this class for a bean archive using beans.xml. Alternatively, @Priority enables it for the application and determines its priority ordering under the CDI rules.

To enable a decorator class:

1. Add a `class` element for the decorator class to the `decorators` element in the `beans.xml` file.
2. In the `class` element, provide the fully qualified class name of the decorator class.

   For classes enabled only through this beans.xml list, ensure that the order of the class elements in the beans.xml file matches the order in which the decorator classes are to be invoked.

> **Note**
>
> Any interceptor classes that are defined for an application are invoked before the application's decorator classes.

For more information about the `beans.xml` file, see Configuring a CDI Application.

Example 9-18 shows a `class` element in the `beans.xml` file for enabling a decorator class.

Example 9-18 Enabling a Decorator Class

This example enables the decorator class `com.example.billpayment.decorator.DataAccessAuthDecorator`.

```
...
<decorators>
     <class>com.example.billpayment.decorator.DataAccessAuthDecorator</class>
</decorators>
...
```

Assigning an EL Name to a CDI Bean Class

EL enables components in the presentation layer to communicate with managed beans that implement application logic.

Components in the presentation layer are typically Jakarta Server Faces (JSF) pages and Jakarta Server Pages (JSP) pages. See JSP Expression Language in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

In the scripting languages in JSP pages and JSF pages, the syntax of an injected variable is identical to the syntax of a built-in variable of these languages. Any CDI bean that is injected into a JSP page or JSF page must be accessible through an EL name. See Giving Beans EL Names in *The Java EE 8 Tutorial* .

To assign an EL name to a CDI bean class, annotate the class declaration of the bean class with the `jakarta.inject.Named` annotation.

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

If you do not specify a name, the EL name is the unqualified class name with the first character in lower case. For example, if the unqualified class name is `ShoppingCart`, the EL name is `shoppingCart`.

To specify a name, set the `value` element of the `@Named` annotation to the name that you require.

> **Note**
>
> A bean can obtain its name from @Named on the class or from a stereotype that supplies @Named. A bean without either naming mechanism has no EL name. Producer methods and fields can also define named beans.

The following example shows how to use the `@Named` annotation to assign an EL name to a CDI bean class. This example assigns the EL name `cart` to the `ShoppingCart` class.

ShoppingCart.java: the returned number is a display-only demonstration value.

```
package com.example.cart;
import jakarta.enterprise.context.SessionScoped;
import jakarta.inject.Named;
import java.io.Serializable;

@SessionScoped
@Named("cart")
public class ShoppingCart implements Serializable {
    private static final long serialVersionUID = 1L;
    public String getTotal() {
        return "0.00";
    }
}
```

Session scope is passivating: bean state, interceptors, decorators and injected dependencies must satisfy the CDI passivation rules. This simple bean has no injected dependencies.

Any bean that a JSP page or JSF page accesses must conform to the JavaBeans standard. To access a CDI managed bean from a JSP page or JSF page through the bean's EL name, use a syntax that is similar to the syntax for JavaBeans components.

The following example shows how an instance of the `ShoppingCart` class is accessed in a JSF page through the EL name that is assigned to the class.

Example 9-19 Accessing a Bean Through its EL Name

This example accesses an instance of the `ShoppingCart` class to display the value of its `total` property in a JSF page.

This property is returned by the `getTotal` getter method of the `ShoppingCart` class.

```
...
<h:outputText value="#{cart.total}"/>
...
```

Defining and Applying Stereotypes

In a large application in which several beans perform similar functions, you may require the same set of annotations to be applied to several bean classes. Defining a stereotype requires you to define the set of annotations only once.

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

Apply the stereotype to reuse its supported metadata and default scope/name rules. An explicit scope or supported naming declaration can override the corresponding default under the CDI rules.

Defining and applying stereotypes involves the tasks that are explained in the following sections:

Defining a Stereotype

A stereotype is an application-defined annotation type that incorporates other annotation types.

To define a stereotype:

1. Define a Java annotation type to represent the stereotype.
2. Annotate the declaration of the annotation type with the following annotations:
   - `jakarta.enterprise.inject.Stereotype`
   - The other annotation types that you want the stereotype to incorporate

     You can specify the following annotation types in a stereotype:
     - A default scope—see Defining the Scope of a Bean
     - `@Alternative`—see Providing Alternative Implementations of a Bean Type
     - One or more interceptor bindings—see Intercepting Method Invocations and Life Cycle Events of Bean Classes
     - `@Named`—see Assigning an EL Name to a CDI Bean Class
3. Specify that the stereotype is to be retained by the virtual machine at run time.

   Use the `java.lang.annotation.Retention(RUNTIME)` meta-annotation for this purpose.
4. Specify that the stereotype may be applied to the program element `TYPE`.

   Use the `java.lang.annotation.Target(TYPE)` meta-annotation for this purpose.

The following example shows the definition of a stereotype.

Example 9-20 Defining a Stereotype

This example defines the stereotype `@Action`, which specifies the following for each bean that the stereotype annotates:

- The default scope is request scope unless the scope is overridden with a scope annotation.
- The default EL name is assigned to the bean unless the name is overridden with the `@Named` annotation.
- The interceptor bindings `@Secure` and `@Transactional` are applied to the bean. The definition of these interceptor bindings is beyond the scope of this example.

```
import jakarta.enterprise.inject.Stereotype;
import jakarta.inject.Named;
import jakarta.enterprise.context.RequestScoped;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@RequestScoped
@Secure
```


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

```
@Transactional
@Named
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action {}
```

Applying Stereotypes to a Bean

To apply stereotypes to a bean, annotate the class declaration of the bean with each stereotype to apply. You can apply any number of stereotypes to a bean. Any stereotype that you apply to a bean must be defined as explained in Defining a Stereotype.

Example 9-21 shows how to apply stereotypes to a bean.

Example 9-21 Applying Stereotypes to a Bean

This example applies the stereotypes `@Action` and `@Mock` to the bean class `MockLoginAction`. The definition of the `@Action` stereotype is shown in Example 9-20. The definition of the `@Mock` stereotype is beyond the scope of this example.

```
@Action
@Mock
public class MockLoginAction extends LoginAction {
 ...
}
```

Using Events for Communications Between Beans

CDI events decouple the bean that fires an event from the beans that observe it. Senders and observers still share the event payload types and qualifiers available to the CDI application. This is in-container event delivery; communication between processes or servers requires an appropriate messaging or remote transport.

At run time, application operations may generate information or state changes that other beans in the CDI application need to observe.

See Using Events in the historical tutorial for background; use the shared payload and in-container delivery contract described above.

Using events for communications between beans involves the tasks that are explained in the following sections:

Defining an Event Type

An event type is a Java class that represents the information that you want to communicate between beans. For example, an event type may represent the state information that a stateful bean must synchronize with state changes in a different tier of an application.

Define an event type for each set of changes that you want to communicate between beans.

To define an event type:

1. Define a Java class to represent the event type.

   Ensure that the class meets these requirements:

   - The class is declared as a concrete Java class.

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

- An event object must have a concrete runtime type with no unresolvable type variables. Observed event types can be supertypes or interfaces under CDI assignability rules; they need not be exactly the event object's declared class.

2. If necessary, define any qualifiers to further distinguish events of this type. For more information, see Defining Qualifiers for Implementations of a Bean Type.
3. Provide code in the class to populate the event payload of event objects that are instantiated from the class.

   The event payload is the information that you want the event to contain. You can use a JavaBeans property with getter and setter methods to represent an item of information in the event payload.

Sending an Event

To communicate a change that occurs in response to an operation, your application must send an event of the correct type when performing the operation. CDI provides a predefined event dispatcher object that enables application code to send an event and select the associated qualifiers at run time.

To send an event:

1. Obtain an instance of the event type to send.
2. Call methods of the event instance to populate the event payload of the event object that you are sending.
3. Inject an instance of the parameterized `jakarta.enterprise.event.Event` interface.

   If you are sending a qualified event, annotate the injection point with the event qualifier.
4. Call the `fire` method of the injected `Event` instance.

   - fire(event) notifies matching synchronous @Observes observers in the firing thread. fireAsync(event) notifies matching @ObservesAsync observers asynchronously and returns a CompletionStage that represents completion; handle exceptional completion as appropriate. It does not invoke ordinary @Observes observers.

Example 9-22 shows how to send an event.

Example 9-22 Sending an Event

This example injects an instance of the event of type `User` with the qualifier `@LoggedIn`. The `fire` method sends only `User` events to which the `@LoggedIn` qualifier is applied.

```
import jakarta.enterprise.event.Event;
import jakarta.enterprise.context.SessionScoped;
import jakarta.inject.Inject;
import java.io.Serializable;

@SessionScoped
public class Login implements Serializable {

    @Inject @LoggedIn Event<User> userLoggedInEvent;
    private User user;

    public void login(Credentials credentials) {

        //... use credentials to find user

        if (user != null) {
            userLoggedInEvent.fire(user);
```


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

```
        }
    }
    ...
}
```

Handling an Event

Any CDI managed bean class can handle events.

To handle an event:

1. In your bean class, define a method to handle the event.
2. In the signature of the method, define a parameter for passing the event to the method.

   Use an observed type assignable from the fired event under CDI rules.
3. Annotate the event parameter with @Observes for synchronous delivery or @ObservesAsync for asynchronous delivery.

   If necessary,

   - set elements of the `@Observes` annotation to specify whether the method is conditional or transactional. See Using Observer Methods to Handle Events in *The Java EE 8 Tutorial* .
   - For synchronous observer ordering, put jakarta.annotation.Priority on the observed event parameter; smaller values run first. Do not rely on portable priority ordering for asynchronous observers.
4. If this observer requires qualifiers, apply those qualifiers to the annotated event parameter.
5. In the method body, provide code for handling the event payload of the event object.

Example 9-23 shows how to declare an observer method for receiving qualified events of a particular type. Example 9-24 shows how to declare an observer method for receiving all events of a particular type.

Example 9-23 Handling a Qualified Event of a Particular Type

This example declares the `afterLogin` method in which the parameter `user` is annotated with the `@Observes` annotation and the `@LoggedIn` qualifier. This method is called when an event of type `User` with the qualifier `@LoggedIn` is sent.

```
import jakarta.enterprise.event.Observes;

    public void afterLogin(@Observes @LoggedIn User user) {
        ...
    }
```

Example 9-24 Handling Any Event of a Particular Type

This example declares the `afterLogin` method in which the parameter `user` is annotated with the `@Observes` annotation. This method is called when any event of type `User` is sent.

```
import jakarta.enterprise.event.Observes;

    public void afterLogin(@Observes User user) {
        ...
    }
```

An observer's qualifiers must be a subset of the event's qualifiers. An observer without an explicit qualifier can receive qualified events of a matching type.

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

The examples below are schematic observer-method fragments; supply the event type, qualifier and enclosing bean.

Injecting a Predefined Bean

The scope and qualifiers of each built-in bean are defined by its integrating specification; do not infer them from resource-injection annotations.

CDI provides predefined beans that implement the following interfaces:

`jakarta.transaction.UserTransaction`
Java Transaction API (JTA) user transaction.

`java.security.Principal`
The abstract notion of a principal, which represents any entity, such as an individual, a corporation, and a login ID.
The principal represents the identity of the current caller. Whenever the injected principal is accessed, it always represents the identity of the current caller.
For example, a principal is injected into a field at initialization. Later, a method that uses the injected principal is called on the object into which the principal was injected. In this situation, the injected principal represents the identity of the current caller when the method is run.

`jakarta.validation.Validator`
Validator for bean instances.
The bean that implements this interface enables a `Validator` object for the default bean validation `ValidatorFactory` object to be injected.

`jakarta.validation.ValidatorFactory`
Factory class for returning initialized `Validator` instances.
The bean that implements this interface enables the default bean validation `ValidatorFactory` object to be injected.

Inject a built-in CDI bean with jakarta.inject.Inject and the required Java type and qualifiers. In Jakarta EE, supported built-in integrations include UserTransaction, Principal and Jakarta Validation's Validator/ValidatorFactory; their availability and use are subject to the applicable component specification. Principal is an interface; use it to access the current caller identity rather than caching a caller name for later requests.

The scope and qualifiers of each built-in bean are defined by its integrating specification; do not infer them from resource-injection annotations.

For more information about injecting resources, see Resource Injection in *The Java EE 8 Tutorial* .

The following complete servlet demonstrates CDI field injection of a built-in UserTransaction.

Example 9-25 Injecting a Predefined Bean

This example injects a user transaction into the servlet class `TransactionServlet`. The user transaction is an instance of the predefined bean that implements the `jakarta.transaction.UserTransaction` interface.

```
package com.example.web;
import jakarta.inject.Inject;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.transaction.UserTransaction;

@WebServlet("/transaction")
```


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

```
public class TransactionServlet extends HttpServlet {
    @Inject
    private UserTransaction transaction;
}
```

@Resource remains a separate Jakarta EE resource-injection mechanism; its plural @Resources form groups class-level declarations and is not used on this field.

Injecting and Qualifying Resources

Java EE 5 resource injection relies on strings for configuration. Typically, these strings are JNDI names that are resolved when an object is created. CDI ensures type-safe injection of beans by selecting the bean class on the basis of the Java type that is specified in the injection point.

A CDI bean can use Jakarta EE resource injection, and producer fields can expose those resource references through type-safe CDI qualifiers. Centralizing declarations is an optional design choice; applications can declare resource injection where appropriate.

1. Optionally centralize related resource declarations in a resource-holder bean.
2. Use producer fields to translate the injected resource type into a CDI bean.

   You can then inject this CDI bean into the application in the same way as any other CDI bean.

   For more information about producer fields, see Defining a Producer Field.

The following example shows how to use Java EE 5 annotations to inject resources.

```
import javax.sql.DataSource;
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.annotation.Resource;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.PersistenceUnit;
import jakarta.ejb.EJB;
import jakarta.xml.ws.WebServiceRef;
...
public class SomeClass {

    @WebServiceRef(lookup="java:app/service/PaymentService")
    PaymentWebService paymentWebService;

    @EJB(beanName="../payment.jar#PaymentService")
    PaymentBusiness paymentBean;

    @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
    DataSource customerDatabase;

    @PersistenceContext(unitName="CustomerDatabase")
    EntityManager customerDatabasePersistenceContext;

    @PersistenceUnit(unitName="CustomerDatabase")
    EntityManagerFactory customerDatabasePersistenceUnit;

    ...
}
```

The following example shows how to inject the same set of resources by combining Java EE 5 resource injection with CDI producer fields.

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

The declaration of the `SomeClass` class is annotated with `@ApplicationScoped` to set the scope of this bean to application. The `@Dependent` scope is implicitly applied to the producer fields.

```
import javax.sql.DataSource;
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import com.example.resources.CustomerDatabase;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.inject.Produces;
import jakarta.annotation.Resource;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.PersistenceUnit;
import jakarta.ejb.EJB;
import jakarta.xml.ws.WebServiceRef;
...
@ApplicationScoped
public class SomeClass {

    @Produces
    @WebServiceRef(lookup="java:app/service/PaymentService")
    PaymentWebService paymentWebService;

    @Produces
    @EJB(beanName="../payment.jar#PaymentService")
    PaymentBusiness paymentBean;

    @Produces @CustomerDatabase
    @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
    DataSource customerDatabase;

    @Produces @CustomerDatabase
    @PersistenceContext(unitName="CustomerDatabase")
    EntityManager customerDatabasePersistenceContext;

    @Produces @CustomerDatabase
    @PersistenceUnit(unitName="CustomerDatabase")
    EntityManagerFactory customerDatabasePersistenceUnit;
    ...
}
```

CDI enables you to use Jakarta EE resources in CDI applications in a way that is consistent with CDI. To use Jakarta EE resources in this way, inject the resources as CDI beans into other beans.

The following example shows how to inject a Jakarta EE resource as a CDI bean into another bean.

This example injects a persistence unit resource into a request-scoped bean.

```
import jakarta.persistence.EntityManagerFactory;
import com.example.resources.CustomerDatabase;
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;

@RequestScoped
public class SomeOtherClass {
    ...
    @Inject @CustomerDatabase
    private EntityManagerFactory emf;
    ...
}
```


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

Another class, for example `YetAnotherClass`, could inject a field of type `SomeOtherClass`. If an instance of `SomeOtherClass` does not already exist in the current request context, CDI performs the following sequence of operations:

1. Constructing the instance of `SomeOtherClass`
2. Injecting the reference to the entity manager factory by using the producer field.
3. Saving the new instance of `SomeOtherClass` in the current request context

In every case, CDI injects the reference to this instance of SomeOtherClass into the field in YetAnotherClass. When the request context ends, CDI destroys SomeOtherClass and releases its dependent contextual references. This does not close or destroy the container-managed EntityManagerFactory or its persistence unit. Do not add a disposer that calls close on these container-managed persistence resources.

Resource-declaration fragments: supply the named EJB module/business interface, generated PaymentWebService client type, persistence unit CustomerDatabase and configured JDBC/ service resources before deployment. These fragments are not a standalone application. The payment types are intentionally distinct; no extra qualifier is needed for those types.

CustomerDatabase.java qualifier shared by the producer and consumer fragments:

```
package com.example.resources;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface CustomerDatabase {}
```

Using CDI With JCA Technology

WebLogic Server supports CDI in embedded resource adapters and global resource adapters. Package a RAR bean archive with META-INF/beans.xml and an explicit discovery mode appropriate to the archive. With annotated discovery, eligible helper bean classes need bean-defining annotations; use all only when full eligible-type discovery is intended.

For more information about the `beans.xml` file, see Configuring a CDI Application.

Jakarta Connectors component classes can receive the supported injection services, but they are not thereby CDI managed beans. Eligible adapter helper classes can be discovered as CDI beans subject to the archive's discovery mode. The following adapter component types are not CDI managed beans:

- **Resource adapter beans.** These beans are classes that are annotated with the `jakarta.resource.spi.Connector` annotation or are declared as corresponding elements in the resource adapter deployment descriptor `ra.xml`.

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

- **Managed connection factory beans.** These beans are classes that are annotated with the `jakarta.resource.spi.ConnectionDefinition` annotation or the `jakarta.resource.spi.ConnectionDefinitions` annotation, or are declared as corresponding elements in `ra.xml`.
- **Activation specification beans.** These beans are classes that are annotated with the `jakarta.resource.spi.Activation` annotation or are declared as corresponding elements in `ra.xml`.
- **Administered object beans.** These beans are classes that are annotated with the `jakarta.resource.spi.AdministeredObject` annotation or are declared as corresponding elements in `ra.xml`.

Configuring a CDI Application

Package eligible classes in a bean archive and choose the discovery mode described below. CDI does not define a separate module packaging format.

CDI Full supports explicit and implicit bean archives. A beans.xml descriptor is not universally required: eligible classes with bean-defining annotations can be discovered in an implicit archive when discovery is enabled. A nonempty current descriptor uses the Jakarta namespace and explicitly states bean-discovery-mode. Use annotated to discover eligible classes with bean-defining annotations, all for full eligible-type discovery in CDI Full, or none to exclude a bean archive. An empty descriptor defaults to annotated, subject to the separately documented WebLogic legacy option.

For migration, AppDeploymentMBean.LegacyEmptyBeansXmlDiscovery defaults to false. When true for an application, qualifying empty beans.xml descriptors, including valid descriptors that omit bean-discovery-mode, use all rather than annotated. An explicit bean-discovery-mode always takes precedence. The option applies when the application is deployed or redeployed; it is separate from ImplicitBeanDiscoveryEnabled and is not a general switch to scan every archive. RAR discovery does not use this compatibility override.

If your CDI application uses alternatives, interceptors, or decorators, enable these items through the applicable beans.xml declaration or supported priority mechanism in the `beans.xml` file. For more information, see:

- Selecting an Alternative Implementation of a Bean Type for an Application
- Enabling an Interceptor
- Enabling a Decorator Class

The required location of the `beans.xml` file depends on the type of the application:

- WAR: WEB-INF/beans.xml or WEB-INF/classes/META-INF/beans.xml; use only one for the same archive. EJB/library/application-client JAR or RAR: META-INF/beans.xml.

Shared bean-archive JARs can be placed in an EAR's library directory, normally lib. Each JAR is discovered/configured according to its own archive metadata and applicable visibility rules.

Example 9-26 shows a `beans.xml` file for configuring a CDI application.

Example 9-26 beans.xml File for Configuring a CDI Application

This example configures a CDI application by enabling the following classes:

- The alternative implementation `com.example.orderprocessor.MockOrderImpl`
- The interceptor class `com.example.billpayment.interceptor.TransactionInterceptor`
- The decorator class `com.example.billpayment.decorator.DataAccessAuthDecorator`

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans 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/beans_4_1.xsd"
       version="4.1" bean-discovery-mode="all">
  <alternatives>
    <class>com.example.orderprocessor.MockOrderImpl</class>
  </alternatives>
  <interceptors>
    <class>com.example.billpayment.interceptor.TransactionInterceptor</class>
  </interceptors>
  <decorators>
    <class>com.example.billpayment.decorator.DataAccessAuthDecorator</class>
  </decorators>
</beans>
```

Enabling and Disabling CDI

CDI for a domain is enabled by default. However, even when an application does not use CDI, there is some CDI initialization that occurs when you deploy an application in WebLogic Server. To maximize deployment performance for applications that do not use CDI, you can disable CDI.

CdiContainerMBean.Policy supplies the domain default, Enabled or Disabled. An explicit weblogic-application.xml cdi-descriptor policy for an EAR, or a supported module descriptor policy for a standalone deployment, takes precedence; otherwise the domain default is used. Disabling CDI prevents CDI processing for the affected deployment. Domain-level changes are broad defaults, so assess applications that depend on CDI before changing them.

The application cdi-descriptor elements are policy and implicit-bean-discovery-enabled. See the application/web descriptor references for supported scope and precedence.

weblogic-application.xml Deployment Descriptor Elements

weblogic.xml descriptor reference

Enabling and Disabling CDI for a Domain

Configure the domain CDI default while preserving the existing domain configuration.

Configuration fragment for an existing domain’s cdi-container element; this is not a complete config.xml. Prefer the WLST procedures below, which preserve the domain configuration. Do not add a duplicate domain or cdi-container element. If editing the file manually, stop the domain first and preserve the rest of its configuration.

```
<cdi-container>
  <policy>Disabled</policy>
</cdi-container>
```

You can use the WLST scripting tool to enable or disable CDI for a domain. The following examples demonstrate how to use WLST to enable and disable CDI for a domain whether you are online or offline.

Example 9-27 Enabling CDI While Online

WLST online configuration edit. Supply the user-config file and protected matching user-key file previously created for the administrative user, and the configured AdminServer URL. Call

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

the function with those three values. Run with no unrelated pending edit; changing these defaults does not reconstruct already-running CDI contexts.

To disable, use Disabled or False as applicable. These examples were checked statically/procedurally, not run against your domain; inspect the printed singleton bean name.

```
def configure_cdi_online(user_config_file, user_key_file, admin_url):
    connect(userConfigFile=user_config_file, userKeyFile=user_key_file,
            url=admin_url)
    edit()
    startEdit()
    cd('/')
    cdi_container = cmo.getCdiContainer()
    print('CDI configuration bean: ' + cdi_container.getName())
    cdi_container.setPolicy('Enabled')
    validate()
    save()
    activate(block='true')
    disconnect()
```

Example 9-28 Enabling CDI While Offline

WLST offline existing-domain edit. Stop the domain before editing it offline. Pass the actual domain directory to the function; it reads and preserves that configuration. It lists the CDI bean names and creates the bean only when absent. Inspect the printed path. Use the online procedure when configuring a running domain.

Use Enabled/Disabled or True/False as applicable to select the intended default.

```
def configure_cdi_offline(domain_directory):
    readDomain(domain_directory)
    cd('/')
    domain_name = get('Name')
    cd('CdiContainer')
    bean_names = ls(returnType='c', returnMap='true')
    if not bean_names:
        cd('/')
        create(domain_name, 'CdiContainer')
        cd('CdiContainer')
        bean_names = ls(returnType='c', returnMap='true')
    if len(bean_names) != 1:
        raise Exception('Expected one CDI configuration bean')
    cd(list(bean_names.keys())[0])
    print(pwd())
    set('Policy', 'Enabled')
    updateDomain()
    closeDomain()
```

Implicit Bean Discovery

An implicit bean archive is discovered under the CDI Full rules, for example from eligible classes with bean-defining annotations or from an empty/annotated beans.xml descriptor. An arbitrary archive without beans.xml is not automatically a bean archive. Archive visibility and extension-specific rules also apply.

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

Disabling implicit discovery and selecting legacy empty-descriptor discovery solve different migration needs. Prefer explicit discovery metadata for predictable new applications; use the compatibility setting when retaining the older descriptor behavior is intentional.

ImplicitBeanDiscoveryEnabled is a boolean, true by default. An explicit application cdi-descriptor value (EAR descriptor, or supported standalone module descriptor) takes precedence over an explicit domain value. Setting false suppresses implicit discovery for affected archives; it does not mean all descriptor-declared CDI archives are disabled. The domain examples below modify the default; use the EAR descriptor or supported standalone module descriptor for narrower configuration.

The application cdi-descriptor elements are policy and implicit-bean-discovery-enabled. See the application/web descriptor references for supported scope and precedence.

weblogic-application.xml Deployment Descriptor Elements

weblogic.xml descriptor reference

Implicit bean archives were introduced in CDI 1.1 / Java EE 7; this is historical context. Use the current annotated/all/none rules and independent switches described in Configuring a CDI Application.

Configuring a CDI Application

Enabling and Disabling Implicit Bean Discovery for a Domain

Configuration fragment for an existing domain's cdi-container element; this is not a complete config.xml. Prefer the WLST procedures below, which preserve the domain configuration. Do not add a duplicate domain or cdi-container element. If editing the file manually, stop the domain first and preserve the rest of its configuration.

```
<cdi-container>
  <implicit-bean-discovery-enabled>false</implicit-bean-discovery-enabled>
</cdi-container>
```

You can use WLST scripting too to enable or disable this feature. The following examples demonstrate how to use WLST to enable and disable implicit bean discovery for a domain whether you are online or offline.

Example 9-29 Enabling Implicit Bean Discovery Using WLST Online

WLST online configuration edit. Supply the user-config file and protected matching user-key file previously created for the administrative user, and the configured AdminServer URL. Call the function with those three values. Run with no unrelated pending edit; changing these defaults does not reconstruct already-running CDI contexts.

To disable, use Disabled or False as applicable. These examples were checked statically/procedurally, not run against your domain; inspect the printed singleton bean name.

```
def configure_implicit_discovery_online(user_config_file, user_key_file, admin_url):
    connect(userConfigFile=user_config_file, userKeyFile=user_key_file,
            url=admin_url)
    edit()
    startEdit()
    cd('/')
    cdi_container = cmo.getCdiContainer()
    print('CDI configuration bean: ' + cdi_container.getName())
    cdi_container.setImplicitBeanDiscoveryEnabled(True)
    validate()
```


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

```
    save()
    activate(block='true')
    disconnect()
```

Example 9-30 Enabling Implicit Bean Discovery Using WLST Offline

WLST offline existing-domain edit. Stop the domain before editing it offline. Pass the actual domain directory to the function; it reads and preserves that configuration. It lists the CDI bean names and creates the bean only when absent. Inspect the printed path. Use the online procedure when configuring a running domain.

Use Enabled/Disabled or True/False as applicable to select the intended default.

```
def configure_implicit_discovery_offline(domain_directory):
    readDomain(domain_directory)
    cd('/')
    domain_name = get('Name')
    cd('CdiContainer')
    bean_names = ls(returnType='c', returnMap='true')
    if not bean_names:
        cd('/')
        create(domain_name, 'CdiContainer')
        cd('CdiContainer')
        bean_names = ls(returnType='c', returnMap='true')
    if len(bean_names) != 1:
        raise Exception('Expected one CDI configuration bean')
    cd(list(bean_names.keys())[0])
    print(pwd())
    set('ImplicitBeanDiscoveryEnabled', True)
    updateDomain()
    closeDomain()
```

Supporting Third-Party Portable Extensions

CDI is intended to be a foundation for frameworks, extensions, and integration with other technologies.

CDI exposes SPIs that enable the development of portable extensions to CDI, such as:

- Integration with business process management engines
- Integration with third-party frameworks such as Spring, Seam, GWT or Wicket
- New technology that is based upon the CDI programming model

The SPIs that enable the development of portable extensions to CDI are provided in the `jakarta.enterprise.inject.spi` package.

Code in CDI extensions can handle events that are sent by the CDI framework.

Portable extensions implement jakarta.enterprise.inject.spi.Extension and are registered through META-INF/services/jakarta.enterprise.inject.spi.Extension. They observe the supported container lifecycle events and run during CDI bootstrap; they are not ordinary application bean observers. CDI also defines build-compatible extensions through jakarta.enterprise.inject.build.compatible.spi, a separate model. See Portable Extensions and Build Compatible Extensions in Jakarta CDI 4.1. Portable Extensions and Build Compatible Extensions in CDI 4.1

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

Using the Built-in Annotation Literals

CDI provides built-in annotation Literal classes for constructing supported annotation instances. The following table lists selected built-in literals in this release.

Syntax

Table 9-2 Built-in Annotation Literals

Classes	Package
`Any`	`jakarta.enterprise.inject`
`Default`	`jakarta.enterprise.inject`
Specializes	`jakarta.enterprise.inject`
Vetoed	`jakarta.enterprise.inject`
`Alternative`	`jakarta.enterprise.inject`
`Typed`	`jakarta.enterprise.inject`
`Nonbinding`	`jakarta.enterprise.util`
`Initialized`	`jakarta.enterprise.context`
BeforeDestroyed	jakarta.enterprise.context
`Destroyed`	`jakarta.enterprise.context`
`RequestScoped`	`jakarta.enterprise.context`
`SessionScoped`	`jakarta.enterprise.context`
`ApplicationScoped`	`jakarta.enterprise.context`
`Dependent`	`jakarta.enterprise.context`
`ConversationScoped`	`jakarta.enterprise.context`

Example 9-31 Built-in Annotation Literals

```
Default defaultLiteral = new Default.Literal();
```

Using the Configurator Interfaces

CDI 2.0 introduced some new configurator interfaces which can be used for dynamically defining or modifying CDI objects.

The configurator interfaces introduced in CDI 2.0 include:

- `AnnotatedTypeConfigurator`
- `InjectionPointConfigurator`
- `BeanAttributesConfigurator`
- `BeanConfigurator`
- `ObserverMethodConfigurator`
- `ProducerConfigurator`

See Using the Configurators Interfaces in *Java EE 8 Tutorial* for more information.

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

The shared package is jakarta.enterprise.inject.spi.configurator; use these interfaces in their supported bootstrap-extension context.

Bootstrapping a CDI Container

For a standalone Java SE application using a CDI implementation, bootstrap through jakarta.enterprise.inject.se.SeContainerInitializer.newInstance(), configure the initializer as needed, then call initialize() to obtain a SeContainer. Close that container when the application finishes, for example with try-with-resources. WebLogic Server initializes CDI for deployed Jakarta EE applications; do not add this standalone bootstrap step inside those applications.

The standard standalone Java SE bootstrap API was introduced in CDI 2.0. The following Java EE 8 tutorial link is historical background.

See Configuring the CDI Container in *Java EE 8 Tutorial* for more information.

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

10

Duplicate - Jakarta JSON Processing

WebLogic Server supports Jakarta JSON Processing by including the JSR-374 reference implementation for use with applications deployed on a WebLogic Server instance. This chapter includes the following sections:

To learn more about JSON concepts, see Jakarta JSON Processing in the *Jakarta EE Tutorial*.

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

11

Jakarta JSON Processing

WebLogic Server supports Jakarta JSON Processing 2.1 (JSON-P) for parsing, generating, querying and transforming JSON data. This release includes the Jakarta JSON Processing API and the Eclipse Parsson implementation. Jakarta JSON Processing 2.1
See Jakarta JSON Processing 2.1.

To learn more about JSON concepts, see Jakarta JSON Processing in the *Jakarta EE Tutorial*.

About JavaScript Object Notation (JSON)

JSON is a lightweight data-interchange format that is widely used as a common format to serialize and deserialize data in applications that communicate with each other over the Internet. These applications are often created using different programming languages and run in very different environments.

JSON is suited to this scenario because it is an open standard, it is easy to read and write, and it is more compact than other representations. RESTful web services typically make extensive use of JSON as the format for the data inside requests and responses, with the JSON representations usually being more compact than the counterpart XML representations since JSON does not have closing tags.

Jakarta JSON Processing provides a convenient way to process (parse, generate, transform, and query) JSON text. For generating and parsing JSON data, there are two programming models, which are similar to those used for XML documents:

- The **object model** creates a tree that represents the JSON data in memory. The tree can then be navigated and analyzed. Although the JSON data created in memory is immutable and cannot be modified, the object model is the most flexible and allows for processing that requires access to the complete contents of the tree. However, it is often slower than the streaming model and requires more memory. The object model generates JSON output by navigating the entire tree at once.

  For information about using the object model, see Object Model API.
- The **streaming model** uses an event-based parser that reads JSON data one element at a time. The parser generates events and stops for processing when an object or an array begins or ends, when it finds a key, or when it finds a value. Each element can be processed or discarded by the application code, and then the parser proceeds to the next event. This approach is adequate for local processing, in which the processing of an element does not require information from the rest of the data. The streaming model generates JSON output to a given stream by making a function call with one element at a time.

  For information about using the streaming model, see Streaming API.

Object Model API

The object model API is a high-level API that provides immutable object models for JSON object and array structures.

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

These JSON structures are represented as object models using the Java types `JsonObject` and `JsonArray`. The interface `jakarta.json.JsonObject` provides a map view to access the unordered collection of zero or more name-value pairs from the model. Similarly, the `jakarta.json.JsonArray` interface provides a list view to access the ordered sequence of zero or more values from the model.

The object model API uses builder patterns to create these object models. The `jakarta.json.JsonObjectBuilder` and `jakarta.json.JsonArrayBuilder` interfaces provide methods to create models of type `JsonObject` and `JsonArray`, respectively.

These object models can also be created from an input source using the `jakarta.json.JsonReader` interface. Similarly, these object models can be written to an output source using the `jakarta.json.JsonWriter` interface.

The following sections show examples of using the object model API:

Creating an Object Model from JSON Data

For a UTF-8 file whose top-level value is an object or array, read the model as follows. The caller handles IOException and parsing failures.

```
import jakarta.json.Json;
import jakarta.json.JsonReader;
import jakarta.json.JsonStructure;
import java.io.IOException;
import java.io.Reader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

public class JsonReadExample {
    public static JsonStructure read(Path file) throws IOException {
        try (Reader input = Files.newBufferedReader(file, StandardCharsets.UTF_8);
             JsonReader reader = Json.createReader(input)) {
            return reader.read();
        }
    }
}


JsonStructure jsonst = JsonReadExample.read(Path.of("jsondata.txt"));
```

Usage from a caller importing java.nio.file.Path and jakarta.json.JsonStructure and handling or declaring IOException:

The object reference `jsonst` can be either of type `JsonObject` or of type `JsonArray`, depending on the contents of the file. `JsonObject` and `JsonArray` are subtypes of `JsonStructure`. This reference represents the top of the tree and can be used to navigate the tree or to write it to a stream as JSON data.

Creating an Object Model from Application Code

The following example shows how to create an object model from application code:

```
import jakarta.json.Json;
import jakarta.json.JsonObject;
...
JsonObject model = Json.createObjectBuilder()
   .add("firstName", "Duke")
```


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

```
   .add("lastName", "Java")
   .add("age", 18)
   .add("streetAddress", "100 Internet Dr")
   .add("city", "JavaTown")
   .add("state", "JA")
   .add("postalCode", "12345")
   .add("phoneNumbers", Json.createArrayBuilder()
      .add(Json.createObjectBuilder()
         .add("type", "mobile")
         .add("number", "111-111-1111"))
      .add(Json.createObjectBuilder()
         .add("type", "home")
         .add("number", "222-222-2222")))
   .build();
```

The object reference model represents the top of the tree, which is created by nesting invocations to the `add` methods and is built by invoking the `build` method. The `jakarta.json.JsonObjectBuilder` interface contains the following `add` methods:

```
JsonObjectBuilder add(String name, BigDecimal value)
JsonObjectBuilder add(String name, BigInteger value)
JsonObjectBuilder add(String name, boolean value)
JsonObjectBuilder add(String name, double value)
JsonObjectBuilder add(String name, int value)
JsonObjectBuilder add(String name, JsonArrayBuilder builder)
JsonObjectBuilder add(String name, JsonObjectBuilder builder)
JsonObjectBuilder add(String name, JsonValue value)
JsonObjectBuilder add(String name, long value)
JsonObjectBuilder add(String name, String value)
JsonObjectBuilder addNull(String name)
```

The `jakarta.json.JsonArrayBuilder` interface contains similar `add` methods that do not have a name (key) parameter. You can nest arrays and objects by passing a new `JsonArrayBuilder` object or a new `JsonObjectBuilder` object to the corresponding `add` method, as shown in this example.

The resulting tree represents the JSON data from JSON Syntax.

Navigating an Object Model

The following example shows a simple approach to navigating an object model:

```
import jakarta.json.JsonValue;
import jakarta.json.JsonObject;
import jakarta.json.JsonArray;
import jakarta.json.JsonNumber;
import jakarta.json.JsonString;
...
public static void navigateTree(JsonValue tree, String key) {
   if (key != null)
      System.out.print("Key " + key + ": ");
   switch(tree.getValueType()) {
      case OBJECT:
         System.out.println("OBJECT");
         JsonObject object = (JsonObject) tree;
         for (String name : object.keySet())
            navigateTree(object.get(name), name);
         break;
      case ARRAY:
         System.out.println("ARRAY");
         JsonArray array = (JsonArray) tree;
```


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

```
            for (JsonValue val : array)
               navigateTree(val, null);
            break;
         case STRING:
            JsonString st = (JsonString) tree;
            System.out.println("STRING " + st.getString());
            break;
         case NUMBER:
            JsonNumber num = (JsonNumber) tree;
            System.out.println("NUMBER " + num.toString());
            break;
         case TRUE:
         case FALSE:
         case NULL:
            System.out.println(tree.getValueType().toString());
            break;
      }
}
```

The `navigateTree` method can be used with the models shown in Creating an Object Model from JSON Data and Creating an Object Model from Application Code as follows:

```
navigateTree(model, null);
```

The `navigateTree` method takes two arguments: a JSON element and a key. The key is used only to help print the key-value pairs inside objects. Elements in a tree are represented by the `JsonValue` type. If the element is an object or an array, a new invocation to this method is made for every element contained in the object or array. If the element is a value, it is printed to standard output.

The `JsonValue.getValueType` method identifies the element as an object, an array, or a value. For objects, the `JsonObject.keySet` method returns a set of strings that contains the keys in the object, and the `JsonObject.get(String name)` method returns the value of the element whose key is `name`. For arrays, `JsonArray` implements the `List<JsonValue>` interface. You can use enhanced `for` loops with the `Set<String>` instance returned by `JsonObject.keySet` and with instances of `JsonArray`, as shown in this example.

The `navigateTree` method for the model shown in Creating an Object Model from Application Code produces the following output:

```
OBJECT
Key firstName: STRING Duke
Key lastName: STRING Java
Key age: NUMBER 18
Key streetAddress: STRING 100 Internet Dr
Key city: STRING JavaTown
Key state: STRING JA
Key postalCode: STRING 12345
Key phoneNumbers: ARRAY
OBJECT
Key type: STRING mobile
Key number: STRING 111-111-1111
OBJECT
Key type: STRING home
Key number: STRING 222-222-2222
```


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

Writing an Object Model to a Stream

The object models created in Creating an Object Model from JSON Data and Creating an Object Model from Application Code can be written to a stream using the `jakarta.json.JsonWriter` interface as follows:

```
import java.io.StringWriter;
import jakarta.json.JsonWriter;
...
StringWriter stWriter = new StringWriter();
JsonWriter jsonWriter = Json.createWriter(stWriter);
jsonWriter.writeObject(model);
jsonWriter.close();

String jsonData = stWriter.toString();
System.out.println(jsonData);

The Json.createWriter method takes an output stream as a parameter. The
JsonWriter.writeObject method writes the object to the stream. The JsonWriter.close
method closes the underlying output stream.

The following example uses try-with-resources to close the JSON writer automatically:

StringWriter stWriter = new StringWriter();
try (JsonWriter jsonWriter = Json.createWriter(stWriter)) {
   jsonWriter.writeObject(model);
}

String jsonData = stWriter.toString();
System.out.println(jsonData);
```

Streaming API

The streaming API is a low-level API designed to process large amounts of JSON data efficiently.

This API consists of the following interfaces:

Interface	Description
jakarta.json.stream.JsonParser	Contains methods to parse JSON in a streaming way. This interface provides forward, read-only access to JSON data using the pull parsing programming model. In this model the application code controls the thread and calls methods in the parser interface to move the parser forward or to obtain JSON data from the current state of the parser.
`jakarta.json.stream.JsonGenerator`	Contains methods to write JSON to an output source in a streaming way. This interface provides methods to write JSON to an output source. The generator writes name-value pairs in JSON objects and values in JSON arrays.

The following sections show examples of using the streaming API:

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

Reading JSON Data Using a Parser

The streaming API can process JSON incrementally without building a complete object model, which can reduce memory use when the application can handle events as they arrive. The following example shows how to create a JsonParser object and how to parse JSON data using events:

Pass the JSON text as jsonData; the parser is closed when processing finishes.

```
import jakarta.json.Json;
import jakarta.json.stream.JsonParser;
import java.io.StringReader;

public class JsonParserExample {
    public static void parse(String jsonData) {
        try (JsonParser parser = Json.createParser(new StringReader(jsonData))) {
            while (parser.hasNext()) {
                JsonParser.Event event = parser.next();
                switch (event) {
                    case KEY_NAME:
                        System.out.print(event + " " + parser.getString() + " - ");
                        break;
                    case VALUE_STRING:
                    case VALUE_NUMBER:
                        System.out.println(event + " " + parser.getString());
                        break;
                    default:
                        System.out.println(event);
                        break;
                }
            }
        }
    }
}

JsonParserExample.parse(JsonWriteExample.write(model));
```

This example consists of three steps:

1. Obtain a parser instance by invoking the `Json.createParser` static method.
2. Iterate over the parser events using the `JsonParser.hasNext` and the `JsonParser.next` methods.
3. Perform local processing for each element.

The example shows the ten possible event types from the parser. The parser's `next` method advances it to the next event.

For the event types `KEY_NAME`, `VALUE_STRING`, and `VALUE_NUMBER`, you can obtain the content of the element by invoking the `JsonParser.getString` method.

For a VALUE_NUMBER event, getInt(), getLong() and getBigDecimal() provide numeric representations. Use isIntegralNumber() to determine whether the current number has an integral representation, and choose an accessor appropriate to the required range and precision. The following is the event output for the object from Creating an Object Model from Application Code. getInt() getLong() getBigDecimal() isIntegralNumber()

```
START_OBJECT
KEY_NAME firstName - VALUE_STRING Duke
```


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

```
KEY_NAME lastName - VALUE_STRING Java
KEY_NAME age - VALUE_NUMBER 18
KEY_NAME streetAddress - VALUE_STRING 100 Internet Dr
KEY_NAME city - VALUE_STRING JavaTown
KEY_NAME state - VALUE_STRING JA
KEY_NAME postalCode - VALUE_STRING 12345
KEY_NAME phoneNumbers - START_ARRAY
START_OBJECT
KEY_NAME type - VALUE_STRING mobile
KEY_NAME number - VALUE_STRING 111-111-1111
END_OBJECT
START_OBJECT
KEY_NAME type - VALUE_STRING home
KEY_NAME number - VALUE_STRING 222-222-2222
END_OBJECT
END_ARRAY
END_OBJECT
```

Writing JSON Data Using a Generator

Call JsonGeneratorExample.write(Path.of("test.txt")) from a caller that imports java.nio.file.Path and handles or declares IOException. This writes UTF-8 JSON and closes the generator and file writer on success or failure.

```
import jakarta.json.Json;
import jakarta.json.stream.JsonGenerator;
import java.io.IOException;
import java.io.Writer;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

public class JsonGeneratorExample {
    public static void write(Path file) throws IOException {
        try (Writer output = Files.newBufferedWriter(file, StandardCharsets.UTF_8);
             JsonGenerator gen = Json.createGenerator(output)) {
            gen.writeStartObject()
               .write("firstName", "Duke")
               .write("lastName", "Java")
               .write("age", 18)
               .write("streetAddress", "100 Internet Dr")
               .write("city", "JavaTown")
               .write("state", "JA")
               .write("postalCode", "12345")
               .writeStartArray("phoneNumbers")
                   .writeStartObject()
                       .write("type", "mobile")
                       .write("number", "111-111-1111")
                   .writeEnd()
                   .writeStartObject()
                       .write("type", "home")
                       .write("number", "222-222-2222")
                   .writeEnd()
               .writeEnd()
            .writeEnd();
        }
    }
}
```

This example obtains a JSON generator by invoking the `Json.createGenerator` static method, which takes a writer or an output stream as a parameter. The example writes JSON data to the

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

`test.txt` file by nesting invocations to the `write`, `writeStartArray`, `writeStartObject`, and `writeEnd` methods. The `JsonGenerator.close` method closes the underlying writer or output stream.

New Features for JSON Processing

The `jakarta.json` API supports new features of JSON Processing such as JSON Pointer, JSON Patch, and JSON Merge Patch. These features can be used to retrieve, transform or manipulate values in an object model.
In this section, the following sample JSON document is used to demonstrate the new features of JSON Processing. This sample contains name-value pairs and the value for the name `"phoneNumbers"` used in this sample, is an array whose elements are two objects.

```
{
   "firstName": "Duke",
   "lastName": "Java",
   "age": 18,
   "streetAddress": "100 Internet Dr",
   "city": "JavaTown",
   "state": "JA",
   "postalCode": "12345",
   "phoneNumbers": [
      { "Mobile": "111-111-1111" },
      { "Home": "222-222-2222" }
   ]
}
```

This section includes the following topics:

JSON Pointer

JSON Pointer defines a string syntax for referencing a location in the target.

A JSON Pointer identifies a location in a JSON document. The Jakarta JsonPointer methods operate on a JsonStructure, such as a JsonObject or JsonArray. An empty pointer string refers to the entire target structure.

Pass the exact sample JSON text from the beginning of this section as jsonString. Call PointerExample.apply(jsonString) to run these operations in order.

```
import jakarta.json.Json;
import jakarta.json.JsonObject;
import jakarta.json.JsonPointer;
import jakarta.json.JsonReader;
import jakarta.json.JsonValue;
import java.io.StringReader;

public class PointerExample {
    public static JsonObject apply(String jsonString) {
        JsonObject contacts;
        try (JsonReader reader = Json.createReader(new
StringReader(jsonString))) {
            contacts = reader.readObject();
        }
        JsonPointer email = Json.createPointer("/email");
        contacts = email.add(contacts, Json.createValue("duke@example.com"));
```


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

```
        JsonPointer name = Json.createPointer("/firstName");
        boolean exists = name.containsValue(contacts);

        JsonPointer phone = Json.createPointer("/phoneNumbers/0");
        JsonValue mobileNumber = phone.getValue(contacts);
        contacts = phone.remove(contacts);

        JsonPointer age = Json.createPointer("/age");
        contacts = age.replace(contacts, Json.createValue(30));
        return contacts;
    }
}
```

exists is true because /firstName exists. mobileNumber is the object {"Mobile":"111-111-1111"}. Every modifying method returns a new structure and its result must be assigned. containsValue tests whether a location exists, not equality with John. JSON object member order is not significant.

The resulting JSON has age 30, the added email and only the Home entry:

```
{
   "firstName": "Duke",
   "lastName": "Java",
   "age": 30,
   "email": "duke@example.com",
   "streetAddress": "100 Internet Dr",
   "city": "JavaTown",
   "state": "JA",
   "postalCode": "12345",
   "phoneNumbers": [
      { "Home": "222-222-2222" }
   ]
}
```

JSON Pointer RFC

JSON Patch

JSON Patch defines a format for expressing a sequence of operations to be applied to a JSON document.

`JsonPatch` mainly consists of two interfaces:

- `JsonPatch` - Provides `apply()`, `toJsonArray()` methods.
- `JsonPatchBuilder` - Provides `add()`, `copy()`, `move()`, `replace()`, `remove()`, and `test()` methods.

Parse the original sample at the start of this section into a JsonObject named contacts. Pass that original object independently to usingBuilder and usingArray; these are two separate demonstrations, not successive patches. No buildPerson helper is required.

Constructing a JSON Patch with JsonPatchBuilder; constructing a JSON Patch from a JsonArray:

```
import jakarta.json.Json;
import jakarta.json.JsonArray;
```


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

```
import jakarta.json.JsonObject;
import jakarta.json.JsonPatch;

public class PatchExamples {
    public static JsonObject usingBuilder(JsonObject contacts) {
        JsonPatch patch = Json.createPatchBuilder()
            .add("/email", "duke@example.com")
            .replace("/age", 30)
            .remove("/phoneNumbers")
            .test("/firstName", "Duke")
            .copy("/lastNameCopy", "/lastName")
            .build();
        return patch.apply(contacts);
    }

    public static JsonArray replacementOperations() {
        return Json.createArrayBuilder()
            .add(Json.createObjectBuilder()
                .add("op", "replace")
                .add("path", "/age")
                .add("value", 30))
            .build();
    }

    public static JsonObject usingArray(JsonObject contacts) {
        JsonPatch patch = Json.createPatch(replacementOperations());
        return patch.apply(contacts);
    }
}
```

usingBuilder adds the email, changes age to 30, removes phoneNumbers, verifies firstName is Duke, and copies lastName to the new root member lastNameCopy. In copy(path, from), the destination path comes first. usingArray changes only age to 30 and preserves both phone entries.

Patch document: [{"op":"replace","path":"/age","value":30}]

JSON Patch RFC

JSON Merge Patch

JSON Merge Patch defines a format and processing rules for applying operations to a JSON document that are based upon specific content of the target document.

`JsonMergePatch` describes changes to be made to a target JSON document using a syntax that closely mimics the document being modified.

Table 11-1 JsonMergePatch syntax

Original	Patch	Result
`{"a":"b"}`	`{"a":"c"}`	`{"a":"c"}`
`{"a":"b"}`	`{"b":"c"}`	`{"a":"b","b":"c"}`
`{"a":"b"}`	`{"a":null}`	`{}`
`{"a":"b","b":"c"}`	`{"a":null}`	`{"b":"c"}`

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

In an object merge patch, a member with a null value removes that member from the target. Arrays are replaced as complete values; their elements are not merged individually. Use JSON Patch when you need explicit operations on individual array elements or to set an object member to JSON null.

You can create a JSON Merge Patch from:

- An existing JSON Merge Patch document represented as a JsonValue`JSONMergePatch`

```
JsonValue contacts = ... ; // The target to be patched
JsonValue patch = ...  ; // JSON Merge Patch
JsonMergePatch mergePatch = Json.createMergePatch(patch);
JsonValue result = mergePatch.apply(contacts);
```

- A difference between two `JsonValues`

```
//The source object
JsonValue source = ... ;
// The modified object
JsonValue target = ... ;
// The diff between source and target in a Json Merge Patch format
JsonMergePatch mergePatch = Json.createMergeDiff(source,target);
```

See JSON Merge Patch RFC.

If you selected to install the Server Examples, the JSON P examples are located in the `ORACLE_HOME\wlserver\samples\server\examples\src\examples\javaee8\jsonp` directory, where `ORACLE_HOME` represents the directory in which you installed WebLogic Server.

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

12

Jakarta JSON Binding

Jakarta JSON Binding (JSON-B) is a standard binding layer for converting Java objects to or from JSON messages. Oracle WebLogic Server 26.1.0.0.0 supports the Jakarta JSON Binding specification by including the JSR-367 reference implementation for use with applications deployed on a WebLogic Server instance.

JSON-B defines a default mapping algorithm for converting existing Java classes to JSON, while enabling developers to customize the mapping process through the use of Java annotations. For more information, see:

- JSON Binding in the *Jakarta EE Tutorial*.
- JSON Binding User Guide
- JSON Binding in the *Jakarta EE Platform*.

This chapter includes the following sections:

About Default Mapping

Default mapping is a set of rules used by the JSON-B engine by default without any customization annotations and custom configuration provided.

This mapping is used for serializing and deserializing basic Java types (such as `java.lang.String`, `java.lang.Long`, and `java.lang.Boolean`), Java SE types (such as `java.math.BigInteger` and `java.util.Optional`), and Java date and time classes.

The main entry point in JSON-B is the jakarta.json.bind.Jsonb interface. It provides a set of overloaded toJson and fromJson methods to serialize Java objects to JSON documents and deserialize them back. Jsonb instances are thread safe and can be reused. It is recommended to have a single instance per configuration type.

Create an instance with JsonbBuilder.create(), or pass a JsonbConfig to JsonbBuilder.create(config). Reuse an application-owned Jsonb while needed, then close it after all users have finished. Do not close a shared instance while another thread is using it.

You can map an object, a collection, or a generic collection:

- Mapping an object

  Use toJson(object) to serialize an object and fromJson(json, Person.class) to deserialize a concrete Person type.

- Mapping a collection or a generic collection

  For a top-level parameterized collection such as List<Person>, pass a java.lang.reflect.Type that retains its element type to fromJson. A raw List.class does not preserve Person; object elements are then mapped as untyped maps. A Type token can be captured from an anonymous subclass as shown below.

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

Run JsonbMappings.main with the JSON-B implementation available. This short-lived example uses try-with-resources; a shared application instance should instead close at its owning lifecycle boundary.

```
import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;

public class JsonbMappings {
    public static class Person {
        public String name;
        public Person() { }
    }

    public static void main(String[] args) throws Exception {
        try (Jsonb jsonb = JsonbBuilder.create()) {
            Person person = new Person();
            person.name = "Duke";
            String json = jsonb.toJson(person);
            Person copy = jsonb.fromJson(json, Person.class);

            Type peopleType = new ArrayList<Person>() { }
                .getClass().getGenericSuperclass();
            List<Person> people = jsonb.fromJson(
                "[{\"name\":\"Duke\"}]", peopleType);

            System.out.println(json);
            System.out.println(copy.name);
            System.out.println(people.get(0).name);
        }
    }
}
```

Expected output:

```
{"name":"Duke"}
Duke
Duke
```

For more information about default mapping, see:

- Jakarta JSON Binding 3.0 specification, §3, Default Mapping

About Customized Mapping

Customize JSON-B mapping declaratively with annotations on your Java types and members, or programmatically with jakarta.json.bind.JsonbConfig. The provider applies both forms during serialization and deserialization. Supply the configuration when creating a Jsonb instance, for example JsonbBuilder.create(config).

JSON-B supports the following customizations:

- Creating custom configurations with formatted output

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

- Changing property names
- Customizing the order of serialized properties
- Ignoring properties
- Changing the default null handling
- Using custom instantiation
- Changing the date and number formats
- Using binary encoding
- Using adapters
- Using serializer and deserializer classes
- Using strict I-JSON support

For more information about customized mapping, see:

- Jakarta JSON Binding 3.0 specification, §4, Customizing Mapping

JSON-B Support in Jakarta REST

In a product that supports Jakarta JSON Binding, implementations must support entity providers for all Java types supported by JSON-B in combination with the media types - `application/json`, `text/json`, and any other media types matching `*/json` or `*/*+json`.

> **Note**
>
> When both JSON-B and JSON-P are available, the JSON-B entity providers take precedence except for jakarta.json.JsonValue and its subtypes, which use the JSON-P providers.

Jakarta REST 4.0, §11.2.6

See Jakarta REST 4.0, §11.2.6, Java API for JSON Binding.

If you selected to install the WebLogic Server Examples, you'll find an example that demonstrates how to use Jakarta JSON Binding with JAX-RS in the `ORACLE_HOME\wlserver\samples\server\examples\src\examples\javaee8\jsonb\jaxrs` 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*.

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

13

Understanding WebLogic Server Application Classloading

Java classloader is a part of the Java virtual machine (JVM) that loads classes into memory. WebLogic Server Jakarta EE application classloading enables WebLogic Server to host multiple isolated applications within the same JVM.

Java Classloading

Classloaders are a fundamental module of the Java language. A classloader is a part of the Java virtual machine (JVM) that loads classes into memory; a classloader is responsible for finding and loading class files at run time. Every successful Java programmer needs to understand classloaders and their behavior.

Java Classloader Hierarchy

The JDK has bootstrap, platform and system class loaders. The bootstrap loader loads core runtime classes, such as java.lang.String. The platform loader loads additional platform classes, and the system loader loads application-path classes. Not every java.* class is defined by the bootstrap loader. A loader's parent is a delegation relationship between loader instances; it is separate from Java class inheritance. The extension-directory mechanism from JDK 8 was removed in JDK 9; do not install application libraries in JAVA_HOME/lib/ext or configure java.ext.dirs. JDK 21 ClassLoader

> **Note**
>
> What Oracle refers to as a "system classpath classloader" is often referred to as the "application classloader" in contexts outside of WebLogic Server. When discussing classloaders in WebLogic Server, Oracle uses the term "system" to differentiate from classloaders related to Jakarta EE applications or libraries (which Oracle refers to as "application classloaders").

Loading a Class

Classloaders use a delegation model when loading a class. The classloader implementation first checks its cache to see if the requested class has already been loaded. This class verification improves performance in that its cached memory copy is used instead of repeated loading of a class from disk. If the class is not found in its cache, the current classloader asks its parent for the class. Only if the parent cannot load the class does the classloader attempt to load the class. If a class exists in both the parent and child classloaders, the parent version is loaded. This delegation model is followed to avoid multiple copies of the same form being loaded. Multiple copies of the same class can lead to a `ClassCastException`.

Classloaders ask their parent classloader to load a class before attempting to load the class themselves. Classloaders in WebLogic Server that are associated with Web applications can be configured to check locally first before asking their parent for the class. This allows Web

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

applications to use their own versions of third-party classes, which might also be used as part of the WebLogic Server product. The prefer-web-inf-classes Element section discusses this in more detail.

prefer-web-inf-classes Element

The `weblogic.xml` Web application deployment descriptor contains a `<prefer-web-inf-classes>` element (a sub-element of the `<container-descriptor>` element). By default, this element is set to `False`. Setting this element to `True` subverts the classloader delegation model so that class definitions from the Web application are loaded in preference to class definitions in higher-level classloaders. This allows a Web application to use its own version of a third-party class, which might also be part of WebLogic Server. See weblogic.xml Deployment Descriptor Elements.

When using this feature, you must be careful not to mix instances created from the Web application's class definition with instances created from the server's definition. If such instances are mixed, a `ClassCastException` results.

Example 13-1 illustrates the `prefer-web-inf-classes` element, its description and default value.

Example 13-1 prefer-web-inf-classes Element

```
/**
* If true, classes located in the WEB-INF directory of a web-app will be
* loaded in preference to classes loaded in the application or system
* classloader.
* @default false
*/
boolean isPreferWebInfClasses();
void setPreferWebInfClasses(boolean b);
```

Changing Classes in a Running Program

WebLogic Server allows you to deploy newer versions of application modules such as EJBs while the server is running. This process is known as hot-deploy or hot-redeploy and is closely related to classloading.

Java classloaders do not have any standard mechanism to undeploy or unload a set of classes, nor can they load new versions of classes. In order to make updates to classes in a running virtual machine, the classloader that loaded the changed classes must be replaced with a new classloader. When a classloader is replaced, all classes that were loaded from that classloader (or any classloaders that are offspring of that classloader) must be reloaded. Any instances of these classes must be re-instantiated.

In WebLogic Server, each application has a hierarchy of classloaders that are offspring of the system classloader. These hierarchies allow applications or parts of applications to be individually reloaded without affecting the rest of the system. WebLogic Server Application Classloading discusses this topic.

Policy Class Loader and Historical Class Caching

The standard WebLogic startup scripts use the Policy Class Loader launcher unless USE_JVM_SYSTEM_LOADER=true selects the JVM system loader. This loader selection is distinct from the historical JDK 8 bulk class-cache feature. The CLASS_CACHE instructions for WebLogic Server 12.1.3 and the 12.2.1 upgrade discussion describe earlier releases; they do

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

not configure AppCDS for this release. For the AppCDS procedure, see Class Caching With Application Class Data Sharing.

Class Caching With Application Class Data Sharing

Class Caching With Application Class Data Sharing

The Application Class Data Sharing (AppCDS) is a class loader optimization that supports archive files of predefined, validated, and linked classes.

This implementation improves the startup time of Oracle WebLogic Server and allows multiple JVMs on the same machine to share memory pages, thereby reducing overall memory usage.

To use this feature, do the following:

1. Generate Class List During WebLogic Server Trial
2. Generate AppCDS Archive
3. Run WebLogic Server With AppCDS Archive

Set DOMAIN_HOME to the actual domain root. Use the same JDK build and matching startup classpath to generate and use an archive. First start in training mode, exercise representative server/application activity, and shut the server down normally to complete the class list. Only then generate the archive and start with useArchive. Repeat training/archive generation after relevant JDK, WebLogic installation or classpath changes.

Generate Class List During WebLogic Server Trial

```
cd "$DOMAIN_HOME/bin"
export APPCDS_CLASS_LIST="$DOMAIN_HOME/WebLogic.classlist"
export APPCDS_ARCHIVE="$DOMAIN_HOME/WebLogic.jsa"
./startWebLogic.sh generateClassList
```

Exercise representative server/application activity, then shut the server down normally before the next step.

Generate AppCDS Archive

```
./generateArchive.sh
```

The two exported absolute paths can be changed consistently; generate the archive only after the training class list is complete.

Run WebLogic Server With AppCDS Archive

```
./startWebLogic.sh useArchive
```

generateClassList and useArchive select the JVM system loader instead of PCL. useArchive uses -Xshare:auto; inspect JVM startup diagnostics to determine whether the archive was used.

WebLogic Server Application Classloading

WebLogic Server classloading is centered on the concept of an application. An application is normally packaged in an Enterprise Archive (EAR) file containing application classes.

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

WebLogic Server application classloading allows WebLogic Server to host multiple isolated applications within the same JVM.

Overview of WebLogic Server Application Classloading

WebLogic Server classloading is centered on the concept of an application. An application is normally packaged in an Enterprise Archive (EAR) file containing application classes. Everything within an EAR file is considered part of the same application. The following may be part of an EAR or can be loaded as standalone applications:

- An EJB JAR file
- A Web application WAR file
- A resource adapter RAR file

> **Note**
>
> See the following sections for more information:
>
> – For information on resource adapters and classloading, see About Resource Adapter Classes.
>
> – For information on overriding generic application files while classloading, see Generic File Loading Overrides in *Deploying Applications to Oracle WebLogic Server*.

If you deploy an EJB and a Web application separately, they are considered two applications. If they are deployed together within an EAR file, they are one application. You deploy modules together in an EAR file for them to be considered part of the same application.

Every application receives its own classloader hierarchy; the parent of this hierarchy is the system classpath classloader. This isolates applications so that application A cannot see the classloaders or classes of application B. In hierarchy classloaders, no sibling or friend concepts exist. Application code only has visibility to classes loaded by the classloader associated with the application (or module) and classes that are loaded by classloaders that are ancestors of the application (or module) classloader. This allows WebLogic Server to host multiple isolated applications within the same JVM.

Application Classloader Hierarchy

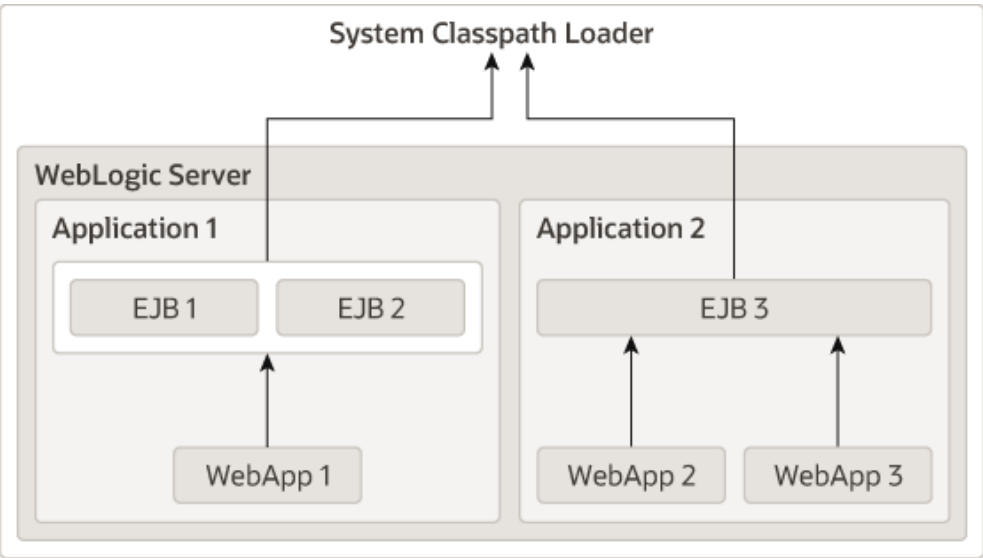
WebLogic Server automatically creates a hierarchy of classloaders when an application is deployed. The root classloader in this hierarchy loads any EJB JAR files in the application. A child classloader is created for each Web application WAR file.

Because it is common for Web applications to call EJBs, the WebLogic Server application classloader architecture allows Jakarta Server Page (JSP) files and servlets to see the EJB interfaces in their parent classloader. This architecture also allows Web applications to be redeployed without redeploying the EJB tier. In practice, it is more common to change JSP files and servlets than to change the EJB tier.

The following graphic illustrates this WebLogic Server application classloading concept.

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

Figure 13-1 WebLogic Server Classloading



If your application includes servlets and JSPs that use EJBs:

- Package the servlets and JSPs in a WAR file
- Package the Jakarta Enterprise Beans (EJB) in an EJB JAR file
- Package the WAR and JAR files in an EAR file
- Deploy the EAR file

Deploying the WAR and EJB module in the same EAR gives them the standard application visibility described here. For a caller in a separate application, package the required client-view interfaces and dependent value types with the caller and use the bean's supported remote client view. A home interface is required only when using the legacy EJB 2.x home/component view. Portable local-view access is guaranteed within the same application; do not assume that a separately deployed WAR can use a local view merely because both deployments run in one JVM. See Application Classloading and Pass-by-Value or Reference.

> **Note**
>
> The Web application classloader contains all classes for the Web application except for the JSP class. The JSP class obtains its own classloader, which is a child of the Web application classloader. This allows JSPs to be individually reloaded.

Custom Module Classloader Hierarchies

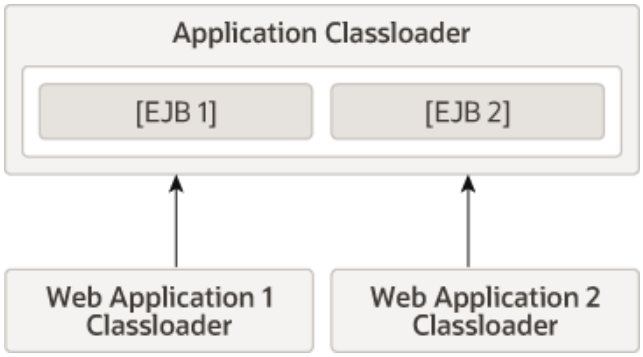
You can create custom classloader hierarchies for an application allowing for better control over class visibility and reloadability. You achieve this by defining a `classloader-structure` element in the `weblogic-application.xml` deployment descriptor file.

The following diagram illustrates how classloaders are organized by default for WebLogic applications. An application level classloader exists where all EJB classes are loaded. For each Web module, there is a separate child classloader for the classes of that module.

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

For simplicity, JSP classloaders are not described in the following diagram.

Figure 13-2 Standard Classloader Hierarchy



This hierarchy is optimal for most applications, because it allows call-by-reference semantics when you invoke EJBs. It also allows Web modules to be independently reloaded without affecting other modules. Further, it allows code running in one of the Web modules to load classes from any of the EJB modules. This is convenient, as it can prevent a Web module from including the interfaces for EJBs that it uses. Note that some of those benefits are not strictly Jakarta EE-compliant.

The invocation benefits depend on the EJB client view and applicable call-by-reference rules; see Application Classloading and Pass-by-Value or Reference.

The ability to create custom module classloaders provides a mechanism to declare alternate classloader organizations that allow the following:

- Reloading individual EJB modules independently
- Reloading groups of modules together
- Reversing the parent child relationship between specific Web modules and EJB modules
- Namespace separation between EJB modules

Declaring the Classloader Hierarchy

You can declare the classloader hierarchy in the WebLogic-specific application deployment descriptor `weblogic-application.xml`.

In the current weblogic-application XML schema, classloader-structure contains zero or more module-ref elements followed by zero or more nested classloader-structure elements. Each module-ref contains one module-uri naming a module in the application.

The top-level element in `weblogic-application.xml` includes an optional `classloader-structure` element. If you do not specify this element, then the standard classloader is used. Also, if you do not include a particular module in the definition, it is assigned a classloader, as in the standard hierarchy. That is, EJB modules are associated with the application root classloader, and Web application modules have their own classloaders.

The `classloader-structure` element allows for the nesting of `classloader-structure` stanzas, so that you can describe an arbitrary hierarchy of classloaders. There is currently a limitation of three levels. The outermost entry indicates the application classloader. For any modules not listed, the standard hierarchy is assumed.

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

> **Note**
>
> JSP classloaders are not included in this definition scheme. JSPs are always loaded into a classloader that is a child of the classloader associated with the Web module to which it belongs.

For more information on these descriptor elements, refer to Enterprise Application Deployment Descriptor Elements.

The following is an example of a classloader declaration (defined in the `classloader-structure` element in `weblogic-application.xml`):

```
<classloader-structure>
   <module-ref>
      <module-uri>ejb1.jar</module-uri>
   </module-ref>
   <module-ref>
      <module-uri>web3.war</module-uri>
   </module-ref>

   <classloader-structure>
      <module-ref>
         <module-uri>web1.war</module-uri>
      </module-ref>
   </classloader-structure>

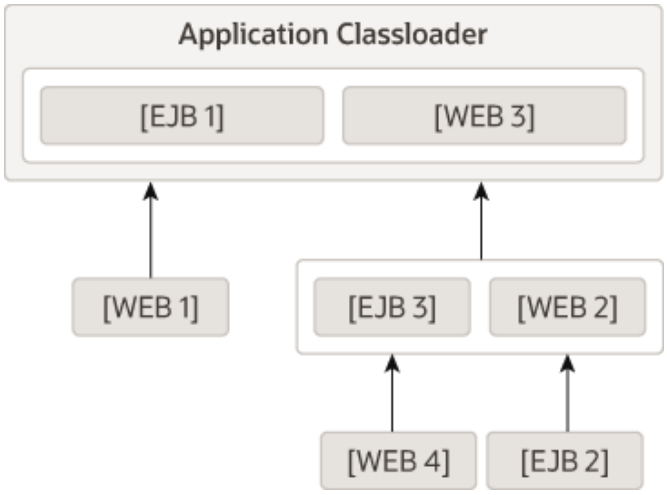
   <classloader-structure>
      <module-ref>
         <module-uri>ejb3.jar</module-uri>
      </module-ref>
      <module-ref>
         <module-uri>web2.war</module-uri>
      </module-ref>

      <classloader-structure>
         <module-ref>
            <module-uri>web4.war</module-uri>
         </module-ref>
      </classloader-structure>
      <classloader-structure>
         <module-ref>
            <module-uri>ejb2.jar</module-uri>
         </module-ref>
      </classloader-structure>
   </classloader-structure>
</classloader-structure>
```

The organization of the nesting indicates the classloader hierarchy. The above stanza leads to a hierarchy shown in the following diagram.

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

Figure 13-3 Example Classloader Hierarchy



User-Defined Classloader Restrictions

User-defined classloader restrictions give you better control over what is reloadable and provide inter-module class visibility. This feature is primarily for developers. It is useful for iterative development, but the reloading aspect of this feature is not recommended for production use, because it is possible to corrupt a running application if an update includes invalid elements. Custom classloader arrangements for namespace separation and class visibility are acceptable for production use. However, programmers should be aware that the Jakarta EE specifications say that applications should not depend on any given classloader organization.

Some classloader hierarchies can cause modules within an application to behave more like modules in two separate applications. For remote-view calls crossing isolated loader branches, copying preserves type isolation. A local-view invocation requires compatible visible types; a hierarchy change does not automatically convert it to a remote invocation.

There are some restrictions to creating user-defined module classloader hierarchies; these are discussed in the following sections.

Servlet Reloading Disabled

If you use a custom classloader hierarchy, servlet reloading is disabled for Web applications in that particular application.

Nesting Depth

Nesting is limited to three levels (including the application classloader). Deeper nestings lead to a deployment exception.

Module Types

Custom classloader hierarchies are currently restricted to Web and EJB modules.

Duplicate Entries

Duplicate entries lead to a deployment exception.

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

Interfaces

The standard WebLogic Server classloader hierarchy makes EJB interfaces available to all modules in the application. Thus other modules can invoke an EJB, even though they do not include the interface classes in their own module. This is possible because EJBs are always loaded into the root classloader and all other modules either share that classloader or have a classloader that is a child of that classloader.

With the custom classloader feature, you can configure a classloader hierarchy so that a callee's classes are not visible to the caller. In this case, the calling module must include the interface classes. This is the same requirement that exists when invoking on modules in a separate application.

Call-by-Value Semantics

Loader placement affects which type definitions callers and callees share. For remote-view calls between isolated branches, values must be adapted to the receiver’s class identity. Local-view calls require a valid shared type arrangement. See Application Classloading and Pass-by-Value or Reference for the invocation contract.

In-Flight Work

During this development-time module/class reloading, the loader switch is not atomic across all modules. In-flight operations can therefore encounter different class versions. Coordinate reloads of callers and callees and avoid using this mechanism to update production applications.

Development-Only Reloading

This warning applies to the incremental reloading described here. Custom classloader arrangements used for namespace separation and visibility can still be used in production. For supported production updates, see Developing Applications for Production Redeployment.

Developing Applications for Production Redeployment

Individual EJB Classloader for Implementation Classes

Legacy EJB implementation-class redeployment can reload an implementation class without replacing the entire module when the supported descriptor model is used and enable-bean-class-redeploy is explicitly enabled in META-INF/weblogic-ejb-jar.xml. This deprecated feature is not supported for EJB 3.x applications. Use module/application redeployment for current EJB applications. A compatible legacy update must not require weblogic.appc recompilation; otherwise redeploy the application.

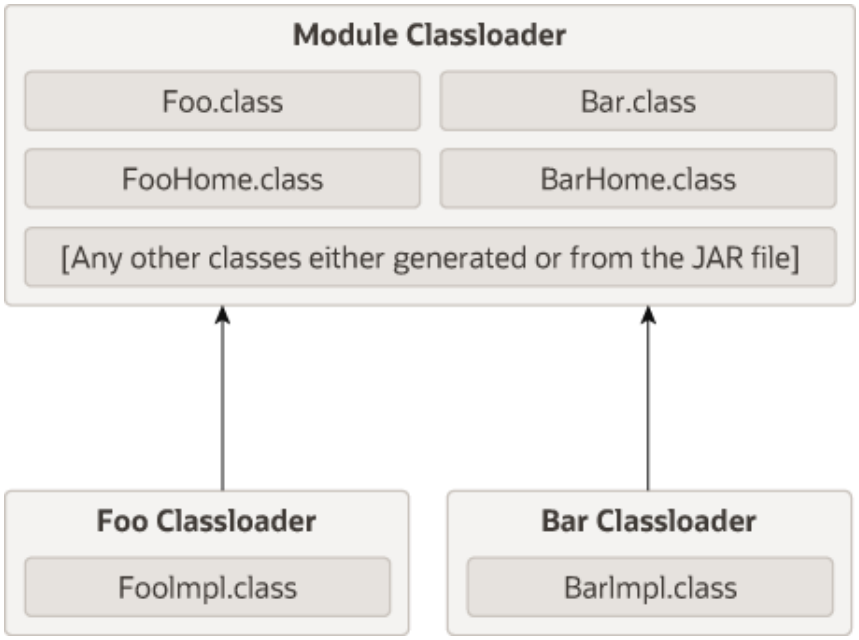
For the explicitly enabled supported legacy feature, the following illustration shows separate implementation loaders for Foo and Bar within one EJB module.

```
<weblogic-ejb-jar xmlns="http://xmlns.oracle.com/weblogic/weblogic-ejb-jar">
  <enable-bean-class-redeploy>true</enable-bean-class-redeploy>
</weblogic-ejb-jar>
```

enable-bean-class-redeploy descriptor reference

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

Figure 13-4 Legacy EJB Implementation-Class Reloading



To perform a partial update of files relative to the root of the exploded application, use the following command line:

Run in an environment configured for WebLogic tools. Set WLS_ADMIN_URL to the configured t3s Administration Server endpoint, WLS_USER_CONFIG to the administrative user's existing protected user-config file, and WLS_USER_KEY to its matching protected key file. Configure trust for that endpoint as required. The deployed exploded application is named myapp; sample module directories mywar, myejb and anotherejb are relative to its root, and myejb contains com/example/FooImpl.class. Replace those sample names with the actual relative paths.

Example 13-2 Performing a Partial File Update

```
java weblogic.Deployer -adminurl "$WLS_ADMIN_URL" \
  -userconfigfile "$WLS_USER_CONFIG" -userkeyfile "$WLS_USER_KEY" \
  -name myapp -redeploy myejb/com/example/FooImpl.class
```

After the `-redeploy` command, you provide a list of files relative to the root of the exploded application that you want to update. This might be the path to a specific element (as above) or a module (or any set of elements and modules). For example:

Example 13-3 Providing a List of Relative Files for Update

```
java weblogic.Deployer -adminurl "$WLS_ADMIN_URL" \
  -userconfigfile "$WLS_USER_CONFIG" -userkeyfile "$WLS_USER_KEY" \
  -name myapp -redeploy mywar myejb/com/example/FooImpl.class anotherejb
```

Given a set of files to be updated, the system tries to figure out the minimum set of things it needs to redeploy. For the supported, explicitly enabled legacy feature, a compatible implementation-only update can reload that implementation. If you specify the whole EJB (in the above example, anotherejb) or if you change and update the EJB home interface, the entire EJB module must be redeployed.

Depending on the classloader hierarchy, this redeployment may lead to other modules being redeployed. Specifically, if other modules share the EJB classloader or are loaded into a

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

classloader that is a child to the EJB's classloader (as in the WebLogic Server standard classloader model) then those modules are also reloaded.

Application Classloading and Pass-by-Value or Reference

Java passes argument values, including reference values, by value. Passing an object reference does not itself copy the object. In EJB terminology, local-view calls share referenced objects, while remote-view calls use copying semantics. WebLogic can optimize eligible in-process remote calls, subject to client-view, class-identity and configuration requirements.

A local EJB view uses shared-object semantics for collocated clients. An eligible in-process remote invocation may avoid copying, but packaging modules in the same EAR is not by itself a guarantee that every remote call uses that optimization. The enable-call-by-reference setting defaults to false; the implementation may recognize safe immutable-only cases.

Application-private copies of a class defined by different loaders are different types. Types defined once by a common ancestor remain shared. Remote calls can preserve isolation by copying values into the receiver’s loading context; the invocation contract is described below.

> **Note**
>
> Package cooperating modules together when they need the same application-visible types. Choose a local or remote client view deliberately; do not infer invocation semantics solely from loader placement.

Using a Filtering Classloader

Under default parent-first delegation, a system-visible definition of com.foo.Baz is found before an application-private definition with the same name. Selective application-package filtering can change that lookup for chosen third-party packages, as described below. A classpath JAR supplies candidate classes; placing it on the path does not eagerly load every class. Keep server and platform API definitions compatible across the application boundary.

The following sections define and describe how to use a filtering classloader:

- What is a Filtering Classloader
- Configuring a Filtering Classloader
- Resource Loading Order

What is a Filtering Classloader

The FilteringClassLoader provides a mechanism for you to configure deployment descriptors to explicitly specify that certain packages should always be loaded from the application, rather than being loaded by the system classloader. This allows you to use alternate versions of applications such as Xerces and Ant. Restrict overrides to application-owned or appropriate third-party implementation packages. Avoid filtering WebLogic server APIs, Jakarta EE APIs in jakarta.* or the Java SE APIs that remain in java.* and javax.*; a blanket namespace filter can break the shared API contracts.

The `FilteringClassLoader` sits between the application classloader and the system classloader. It is a child of the system classloader and the parent of the application classloader. The `FilteringClassLoader` intercepts the `loadClass(String` *`className`*`)` method and compares the *`className`* with a list of packages specified in `weblogic-application.xml` file. If

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

the package matches the *className*, the FilteringClassLoader throws a ClassNotFoundException. This exception notifies the application classloader to load this class from the application.

Configuring a Filtering Classloader

To configure the FilteringClassLoader to specify that a certain package is loaded from an application, add a prefer-application-packages descriptor element to weblogic-application.xml which details the list of packages to be loaded from the application. The following example specifies that org.apache.log4j.* and antlr.* packages are loaded from the application, not the system classloader:

```
<prefer-application-packages>
  <package-name>org.apache.log4j.*</package-name>
  <package-name>antlr.*</package-name>
</prefer-application-packages>
```

The prefer-application-packages descriptor element can also be defined in weblogic.xml. See prefer-application-packages.

Replace com.example.thirdparty.* with the actual bundled implementation package.

```
<container-descriptor>
  <prefer-web-inf-classes>false</prefer-web-inf-classes>
  <prefer-application-packages>
    <package-name>com.example.thirdparty.*</package-name>
  </prefer-application-packages>
</container-descriptor>
```

For selective prefer-application-packages or prefer-application-resources in weblogic.xml, prefer-web-inf-classes must be false. Do not combine those selective settings with prefer-web-inf-classes=true.

You can specify that a certain package be loaded for a WAR file included within an EAR file by configuring the FilteringClassLoader in the weblogic.xml file of the WAR file.

For example, A.ear contains B.war. A.ear defines the FilteringClassLoader in weblogic-application.xml, and B.war defines a different FilteringClassLoader in weblogic.xml. When you deploy A.ear, B.war loads the package defined in the FilteringClassLoader in weblogic.xml. The WAR-level FilteringClassLoader has priority over the EAR-level FilteringClassLoader for this WAR file.

For aid in configuring filtering classloaders, see Using the Classloader Analysis Tool (CAT).

Resource Loading Order

The resource loading order is the order in which java.lang.ClassLoader methods getResource()and getResources() return resources. When filtering is enabled, this order is slightly different from the case when filtering is disabled. Filtering is enabled implies that there are one or more package patterns in the FilteringClassLoader. Without any filtering (default), the resources are collected in the top-down order of the classloader tree. For instance, if Web (1) requests resources, the resources are grouped in the following order: Sys (3), App (2) and Web(1). See Example 13-4.

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

> **Note**
>
> The resources are returned in the default Java EE delegation model beneath the `FilteringClassLoader`. Only the resources from the parent of the `FilteringClassLoader` are appended to the end of the enumeration being returned.

Example 13-4 Using the System Classloader

These examples assume ordinary parent-first module loading and no explicit prefer-application-resources patterns. With no package filter, getResources returns System, App, Web. With the illustrated package filter, it returns App, Web, System. Explicit resource preferences can suppress the parent resources for matching names.

```
System (3)
   |
  App (2)
   |
  Web (1)
```

To be more explicit, given a resource `META-INF/foo.xml` which exists in all the classloaders, would return the following list of URLs:

```
META-INF/foo.xml - from the System ClassLoader (3)
META-INF/foo.xml - from the App ClassLoader (2)
META-INF/foo.xml - from the Web ClassLoader (1)
```

When filtering is enabled, the resources from the child of the `FilteringClassLoader` (an application classloader) down to the calling classloader are returned before the ones from the system classloader. In Example 13-5, if the same resource existed in all the classloaders (D), (B) and (A) one would get them in the following order if requested by the Web classloader:

```
META-INF/foo.xml - from the App ClassLoader (B)
META-INF/foo.xml - from the Web ClassLoader (A)
META-INF/foo.xml - from the System ClassLoader (D)
```

Example 13-5 Using a Filtering Classloading Implementation

```
System (D)
  |
  FilteringClassLoader (filterList := x.y.*) (C)
   |
  App (B)
   |
  Web (A)
```

If the application classloader requested the same resource, the following order would be obtained.

```
META-INF/foo.xml - from the App ClassLoader (B)
META-INF/foo.xml - from the System ClassLoader (D)
```

getResource(name) returns one matching resource URL, or null when none is found. In this illustrated hierarchy it returns the first resource in the shown order; getResourceAsStream(name) opens that resource as an InputStream.

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

Resolving Class References Between Modules and Applications

WebLogic Server deploys applications in separate classloaders to maintain independence and to facilitate dynamic redeployment and undeployment. Because of this, you need to package your application classes in such a way that each module has access to the classes it depends on.

Your applications may use many different Java classes, including Enterprise Beans, servlets and Jakarta Pages, utility classes, and third-party packages. In some cases, you may have to include a set of classes in more than one application or module. This section describes how WebLogic Server uses multiple classloaders so that you can stage your applications successfully.

For more information about analyzing and resolving classloading issues, see Using the Classloader Analysis Tool (CAT).

About Resource Adapter Classes

Resource-adapter visibility depends on deployment scope and configuration. A standalone adapter normally has its own loader. For an embedded adapter without a custom arrangement or global-class-access override, EnableEECompliantClassloadingForEmbeddedAdapters controls whether it uses the EAR application loader or an adapter-specific loader; its derived default is true for domains whose version is 12.2.1 or later. An explicit classloader-structure can supply the adapter loader. The adapter's enable-global-access-to-classes option takes precedence over that custom structure. Package shared client API/value classes once at a common application-visible location when appropriate; do not duplicate the adapter implementation indiscriminately in APP-INF and module archives to work around an assumed isolation rule.

Packaging Shared Utility Classes

WebLogic Server provides a location within an EAR file where you can store shared utility classes. Place utility JAR files in the `APP-INF/lib` directory and individual classes in the `APP-INF/classes` directory. (Do not place JAR files in the `/classes` directory or classes in the `/lib` directory.) These classes are loaded into the root classloader for the application.

This feature obviates the need to place utility classes in the system classpath or place classes in an EJB JAR file (which depends on the standard WebLogic Server classloader hierarchy). Be aware that using this feature is subtly different from using the manifest `Class-Path` described in the following section. With this feature, class definitions are shared across the application. With manifest `Class-Path`, the classpath of the referencing module is simply extended, which means that separate copies of the classes exist for each module.

Manifest Class-Path

The Jakarta EE specification provides the manifest `Class-Path` entry as a means for a module to specify that it requires an auxiliary JAR of classes. You only need to use this manifest `Class-Path` entry if you have additional supporting JAR files as part of your EJB JAR or WAR file. In such cases, when you create the JAR or WAR file, you must include a manifest file with a `Class-Path` element that references the required JAR files.

The following is a simple manifest file that references a `utility.jar` file:

```
Manifest-Version: 1.0 [CRLF]
Class-Path: utility.jar [CRLF]
```


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

In the first line of the manifest file, you must always include the `Manifest-Version` attribute, followed by a new line (CR | LF |CRLF) and then the `Class-Path` attribute. More information about the manifest format can be found at: https://docs.oracle.com/en/java/javase/17/docs/specs/jar/jar.html

The manifest `Class-Path` entries refer to other archives relative to the current archive in which these entries are defined. This structure allows multiple WAR files and EJB JAR files to share a common library JAR. For example, if a WAR file contains a manifest entry of `y.jars`, this entry should be next to the WAR file (not within it) as follows:

```
/<directory>/x.war
/<directory>/y.jars
```

The manifest file itself should be located in the archive at `META-INF/MANIFEST.MF`.

See http://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html.

Using the Classloader Analysis Tool (CAT)

CAT is a web-based class analysis tool that simplifies filtering classloader configuration and aids you in analyzing classloading issues, such as detecting conflicts, debugging application classpaths and class conflicts, and proposes solutions to help you resolve them.

CAT is a standalone web application, distributed as a single WAR file, `wls-cat.war`, exposing its features through a web-based front end.

Use the CAT application supplied with the same WebLogic Server release as the target server. You can add it to the domain with ORACLE_HOME/wlserver/common/templates/wls/wls_cat.jar, or deploy the matching WL_HOME/server/lib/wls-cat.war as a web application to the server whose applications you need to inspect. The template installs the CAT_Application deployment; do not deploy a duplicate copy after applying it.

> **Note**
>
> Run CAT on a JDK certified for this WebLogic Server release. These instructions do not establish compatibility of the current CAT WAR with earlier WebLogic releases.

Opening the CAT Interface

CAT has a simple web GUI that displays all your currently running applications and modules.

To begin using CAT, open your browser to `http://wls-host:port/wls-cat/` and then enter your console login credentials.

In the navigation pane, select the application or module that you want to analyze; a brief description of it is shown in the right-side pane. Use the right-side pane to perform actions and analyses on the selected application or module, such as:

- Analyze classloading conflicts
- View the system and application classloaders
- Generate reports

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

How CAT Analyzes Classes

CAT analyzes classes loaded by the system classpath classloader and the WebLogic Server main application classloaders, defined here as the filtering, application, and module classloaders. You can perform analysis at the class, package, or JAR level. The results for each action you select can be shown in either a basic view or a detailed view.

Here are some of the tasks which you can perform using CAT:

- Display basic information about applications and modules
- Analyze classloading conflicts
- Review proposed solutions
- Get suggestions for configuring filtering classloaders
- Display the classloader hierarchy and the entire classpath for each classloader
- Search for a class (or a resource) on a classloader

Identifying Class References through Manifest Hierarchies

Applications can have multiple manifest references to classes that are not directly present in the application's classpath, but which are *chained* into the Classpath by manifest references. In some cases, application developers may not be aware that additional classes have been unknowingly pulled into the application's classpath from other JARs, which in turn have manifest references to other JARs.

CAT has the ability to search through an application's or module's classpath to detect and display the underlying chained manifest references, as shown in the following Sample EAR with Manifest Hierarchies example:

```
cat4mf.ear
  +- ejb.jar
  +- web-mf-in-root.war
  +- lib
      +- applib.jar
  +- apputil_1.jar
  +- apputil_1_1.jar
  +- apputil_1_1_1.jar
  +- apputil_1_2.jar
  +- apputil_1_2_1.jar
  +- ejbutil_1.jar
  +- ejbutil_1_1.jar
  +- ejbutil_1_2.jar
  +- ejbutil_1_2_1.jar
  +- webutil_1.jar
  +- webutil_1_1.jar
  +- webutil_1_1_1.jar
  +- webutil_2.jar
  +- webutil_2_1.jar
```

The `ejb.jar` has a manifest reference to `ejbutil_1.jar`, which has references to both `ejbutil_1_1.jar` and `ejbutil_1_2.jar`, which has a further reference to `ejbutil_1_2_1.jar`, as follows:

```
  ejb.jar
     -> ejbutil_1.jar
            -> ejbutil_1_1.jar
```


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

```
-> ejbutil_1_2.jar
        -> ejbutil_1_2_1.jar
```

Using CAT to Display the Manifest References

1. Open the CAT tool, as described in Opening the CAT Interface.
2. Use the navigation pane to select the running application or module to analyze.

   Note: The manifest references can best be analyzed from the module level rather than the application level.
3. In the **Summary for Application** pane, click the **Classloader Tree** view to list all the classloaders for the selected application/module.
   - Selecting the **detailed** view from the menu displays the classpath of each classloader.
   - The hash code of each classloader is an active URL.
4. Click the classloader hash code URL you want to analyze.
5. The **Classloader** page defaults to the basic view, so select the **detailed** view to see the classpath and the classes loaded by the classloader.
6. Enter one of the loaded classnames in the Resource to analyze field (using the format pckgname.classname), and click Analyze Resource.
7. The **Manifest References** section of the detailed output provides the list of chained manifest references for the selected classname.

   Continuing with Sample EAR with Manifest Hierarchies example, the output should look like this:

```
path/to/user_projects/applications/cat4mf/y79s0z/ejb.jar
   path/to/user_projects/applications/cat4mf/y79s0z/ejbutil_1.jar
      path/to/user_projects/applications/cat4mf/y79s0z/ejbutil_1_2.jar
        path/to/user_projects/applications/cat4mf/y79s0z/ejbutil_1_2_1.jar
```

Sharing Applications and Modules By Using Jakarta EE Libraries

Jakarta EE libraries provide an easy way to share one or more different types of Jakarta EE modules among multiple enterprise applications.

A Jakarta EE library is a single module or collection of modules that is registered with the Jakarta EE application container upon deployment. For more information, see Creating Shared Jakarta EE Libraries and Optional Packages.

Adding JARs to the Domain `/lib` Directory

WebLogic Server includes a `lib` subdirectory, located in the domain directory, that you can use to add one or more JAR files, so that the JAR file classes are available within a separate system level classloader to all Jakarta EE applications running on WebLogic Server instances in the domain.

The JARS in the domain `/lib` directory will not be appended to the system classpath. The classloader that gets created is a child of the system classloader. Any classes that are in JARs in the domain `/lib` directory will only be visible to Jakarta EE applications, such as EAR files. Classes in the system classpath cannot access classes in the domain `/lib` directory.

The `lib` subdirectory is intended for JAR files that change infrequently and are required by all or most applications deployed in the server. For example, you might use the `lib` directory to store third-party utility classes that are required by all Jakarta EE deployments in a domain.

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

Third-party utility classes will be made available because the domain `/lib` classloader will be the parent of any Jakarta EE application.

The `lib` directory is not recommended as a general-purpose method for sharing JAR files between one or two applications deployed in a domain, or for sharing JARs that need to be updated periodically. If you update a JAR in the `lib` directory, you must reboot all servers in the domain in order for applications to realize the change. If you need to share a JAR file or Jakarta EE modules among several applications, use the Jakarta EE libraries feature described in Creating Shared Jakarta EE Libraries and Optional Packages.

To share JARs using the `lib` directory:

1. Shutdown all servers in the domain.
2. Copy the JAR file(s) to share into a `lib` subdirectory of the domain directory. For example:

   Set DOMAIN_HOME to the actual domain directory and replace /path/to/utility.jar with the source JAR path. Repeat the copy into each Managed Server machine's local DOMAIN_HOME/lib when it does not share the Administration Server's domain directory.

   ```
   mkdir -p "$DOMAIN_HOME/lib"
   cp "/path/to/utility.jar" "$DOMAIN_HOME/lib/"
   ```

   > **Note**
   >
   > WebLogic Server must have read access to the `lib` directory during startup.
   >
   > The Administration Server does not automatically copy files in the lib directory to Managed Servers on remote machines. If you have Managed Servers that do not share the same physical domain directory as the Administration Server, you must manually copy JAR file(s) to the DOMAIN_HOME/lib directory on the Managed Server machines.

3. Start the Administration Server and all Managed Servers in the domain.

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

14

Creating Shared Jakarta EE Libraries and Optional Packages

You can share components and classes among applications using shared Jakarta EE libraries supported in WebLogic Server.

Overview of Shared Jakarta EE Libraries and Optional Packages

The shared Jakarta EE library feature in WebLogic Server provides an easy way to share one or more different types of Jakarta EE modules among multiple enterprise applications. A shared Jakarta EE libraries can be referenced by enterprise applications and you can also create libraries that can be referenced only by another Web application.

A shared Jakarta EE library is a single module or collection of modules that is registered with the Jakarta EE application container upon deployment. A shared Jakarta EE library can be any of the following:

- Standalone EJB module
- Standalone Web application module
- Multiple EJB modules packaged in an enterprise application
- Multiple Web application modules package in an enterprise application
- Single plain JAR file

Oracle recommends that you package a shared Jakarta EE library into its appropriate archive file (EAR, JAR, or WAR). However, for development purposes, you may choose to deploy shared Jakarta EE libraries as exploded archive directories to facilitate repeated updates and redeployments.

After the shared Jakarta EE library has been registered, you can deploy enterprise applications that reference the library. Each referencing application receives a reference to the required library on deployment, and can use the modules that make up the library as if they were packaged as part of the referencing application itself. The library classes are added to the class path of the referencing application, and the primary deployment descriptors of the referencing application or module are merged (in memory) with those of the modules that make up the shared Jakarta EE library.

In general, this topic discusses shared Jakarta EE libraries that can be referenced only by enterprise applications. You can also create libraries that can be referenced only by another Web application. The functionality is very similar to application libraries, although the method of referencing them is slightly different. See Web Application Shared Jakarta EE Library Information for details.

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

> **Note**
>
> WebLogic Server also provides a simple way to add one or more JAR files to the WebLogic Server System classpath, using the `lib` subdirectory of the domain directory. See Adding JARs to the Domain /lib Directory.

Library Directories

The Jakarta EE platform provides several mechanisms for applications to use optional packages and shared libraries. Libraries can be bundled with an application or may be installed separately for use by any application. An EAR file may contain a directory that contains libraries packaged in JAR files. The `library-directory` element of the EAR file's deployment descriptor contains the name of this directory. If a `library-directory` element isn't specified, or if the EAR file does not contain a deployment descriptor, the directory named `lib` is used. An empty `library-directory` element may be used to specify that there is no library directory. All files in this directory (but not in subdirectories) with a `.jar` extension must be made available to all components packaged in the EAR file, including application clients. These libraries may reference other libraries, either bundled with the application or installed separately.

This feature is similar to the `APP-INF/lib` feature supported in WebLogic Server. If both `APP-INF/lib` and `library-directory` exist, then the jars in the `library-directory` would take precedence; that is, they would be placed before the `APP-INF/lib` jar files in the classpath. For more information on `APP-INF/lib`, see Resolving Class References Between Modules and Applications and Organizing Shared Classes in a Split Development Directory.

Versioning Support for Libraries

WebLogic Server supports specification and implementation versioning for registered libraries. The manifest names have historical roots in Optional Package Versioning; the WebLogic version-selection rules for this release are given below. Historical background: `https://docs.oracle.com/javase/8/docs/technotes/guides/extensions/versioning.html`

- Specification Version—Identifies the version number of the specification (for example, the Jakarta EE specification version) to which a shared Jakarta EE library or optional package conforms.
- Implementation Version—Identifies the version number of the actual code implementation for the library or package. For example, this would correspond to the actual revision number or release number of your code. Note that you must also provide a specification version in order to specify an implementation version.

As a best practice, Oracle recommends that you always include version information (a specification version, or both an implementation and specification version) when creating shared Jakarta EE libraries. Creating and updating version information as you develop shared components allows you to deploy multiple versions of those components simultaneously for testing. If you include no version information, or fail to increment the version string, then you must undeploy existing libraries before you can deploy the newer one. See Deploying Shared Jakarta EE Libraries and Dependent Applications.

Versioning information in the referencing application determines the library and package version requirements for that application. Different applications can require different versions of a given library or package. For example, a production application may require a specific version of a library, because only that library has been fully approved for production use. An internal application may be configured to always use a minimum version of the same library.

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

Applications that require no specific version can be configured to use the latest version of the library. Referencing Shared Jakarta EE Libraries in an Enterprise Application.

Shared Jakarta EE Libraries and Optional Packages Compared

Optional packages and shared Jakarta EE libraries have the following features in common:

- Both are registered with WebLogic Server instances at deployment time.
- Both support an optional implementation version and specification version string.
- Applications that reference shared Jakarta EE libraries and optional packages can specify required versions for the shared files.
- Optional packages can reference other optional packages, and shared Jakarta EE libraries can reference other shared Jakarta EE libraries.

Optional packages differ from shared Jakarta EE Libraries in the following basic ways:

- Optional packages are plain JAR files, whereas shared Jakarta EE libraries can be plain JAR files, Jakarta EE enterprise applications, or standalone Jakarta EE modules (EJB and Web applications). This means that libraries can have valid Jakarta EE and WebLogic Server deployment descriptors. Any deployment descriptors in an optional package JAR file are ignored.
- Any Jakarta EE application or module can reference an optional package (using `META-INF/MANIFEST.MF`), whereas only enterprise applications and Web applications can reference a shared Jakarta EE library (using `weblogic-application.xml` or `weblogic.xml`).

In general, use shared Jakarta EE libraries when you need to share one or more EJB, Web application or enterprise application modules among different enterprise applications. Use optional packages when you need to share one or more classes (packaged in a JAR file) among different Jakarta EE modules.

Use WebLogic registered optional packages for manifest-declared dependencies on server-registered JARs. For portable libraries bundled with an application, use the Jakarta EE mechanisms described in Library Directories and Manifest Class-Path. Use WebLogic shared libraries when separately registered modules or JARs need the library-ref deployment model.

> **Note**
>
> Oracle documentation and WebLogic Server utilities use the term *shared Jakarta EE library* to refer to both libraries and optional packages. Optional packages are called out only when necessary.

Additional Information

For information about deploying and managing shared Jakarta EE libraries, optional packages, and referencing applications from the administrator's perspective, see Deploying Shared Jakarta EE Libraries and Dependent Applications in *Deploying Applications to Oracle WebLogic Server*.

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

Creating Shared Jakarta EE Libraries

You can deploy the Jakarta EE modules such as an EJB, a Web application, an enterprise application, a plain Java class, and others as a shared Jakarta EE library. These modules can be shared among multiple enterprise applications in WebLogic Server.

Create a shared library from an EAR, EJB JAR, WAR, or plain library JAR using the packaging appropriate to its contents.

1. For an EAR, EJB or WAR library, assemble a valid module/application and include the descriptors required by its metadata and configuration. For a plain JAR library or optional package, assemble the compiled classes in their package directories. See Assembling Shared Jakarta EE Library Files or Assembling Optional Package Class Files, as appropriate.
2. Create and edit the `MANIFEST.MF` file for the shared Jakarta EE library to specify the name and version string information.

   See Editing Manifest Attributes for Shared Jakarta EE Libraries.
3. Package the shared Jakarta EE library for distribution and deployment.

   See Packaging Shared Jakarta EE Libraries for Distribution and Deployment.

Assembling Shared Jakarta EE Library Files

The following types of Jakarta EE modules can be deployed as a shared Jakarta EE library:

- An EJB module, either an exploded directory or packaged in a JAR file.
- A Web application module, either an exploded directory or packaged in a WAR file.
- An enterprise application, either an exploded directory or packaged in an EAR file.
- A plain Java class or classes packaged in a JAR file.
- A shared Jakarta EE library referenced from another library. (See Web Application Shared Jakarta EE Library Information.)

Shared Jakarta EE libraries have the following restrictions:

- You must ensure that context roots in Web application modules of the shared Jakarta EE library do not conflict with context roots in the referencing enterprise application. If necessary, you can configure referencing applications to override a library's context root. See Referencing Shared Jakarta EE Libraries in an Enterprise Application.
- Shared Jakarta EE libraries cannot be nested. For example, if you are deploying an EAR as a shared Jakarta EE library, the entire EAR must be designated as the library. You cannot designate individual Jakarta EE modules within the EAR as separate, named libraries.
- Target the library to the servers or clusters that will host its consumers. Include valid standard or WebLogic descriptors when required by the module type or selected configuration; use a deployment plan when needed. A plain library JAR or optional package does not need application/module deployment descriptors.

Oracle recommends packaging shared Jakarta EE libraries as enterprise applications, rather than as standalone Jakarta EE modules. This is because the URI of a standalone module is derived from the deployment name, which can change depending on how the module is deployed. By default, WebLogic Server uses the deployment archive filename or exploded archive directory name as the deployment name. If you redeploy a standalone shared Jakarta

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

EE library from a different file or location, the deployment name and URI also change, and referencing applications that use the wrong URI cannot access the deployed library.

If you choose to deploy a shared Jakarta EE library as a standalone Jakarta EE module, always specify a known deployment name during deployment and use that name as the URI in referencing applications.

Assembling Optional Package Class Files

Any set of classes can be organized into an optional package file. The collection of shared classes will eventually be packaged into a standard JAR archive. However, because you will need to edit the manifest file for the JAR, begin by assembling all class files into a working directory:

1. Create a working directory for the new optional package. For example:

   ```
   mkdir /apps/myOptPkg
   ```

2. Copy the compiled class files into the working directory, creating the appropriate package subdirectories as necessary. For example:

   The sample class package is org.myOrg.myProduct. Replace the root paths with your actual work and compiled-class directories.

   ```
   OPT_PKG_WORK="/apps/myOptPkg"
   OPT_PKG_CLASSES="/build/classes/myOptPkg"
   mkdir -p "$OPT_PKG_WORK/org/myOrg/myProduct"
   cp "$OPT_PKG_CLASSES"/org/myOrg/myProduct/*.class \
     "$OPT_PKG_WORK/org/myOrg/myProduct/"
   ```

3. If you already have a JAR file that you want to use as an optional package, extract its contents into the working directory so that you can edit the manifest file:

   ```
   cd /apps/myOptPkg
   jar xvf /build/libraries/myLib.jar
   ```

Editing Manifest Attributes for Shared Jakarta EE Libraries

The name and version information for a shared Jakarta EE library are specified in the `META-INF/MANIFEST.MF` file. Table 14-1 describes the valid shared Jakarta EE library manifest attributes.

Table 14-1 Manifest Attributes for Jakarta EE Libraries

Attribute	Description
`Extension-Name`	An optional string value that identifies the name of the shared Jakarta EE library. Referencing applications must use the exact `Extension-Name` value to use the library. As a best practice, always specify an `Extension-Name` value for each library. If you do not specify an extension name, one is derived from the deployment name of the library. Default deployment names are different for archive and exploded archive deployments, and they can be set to arbitrary values in the deployment command.

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

Table 14-1 (Cont.) Manifest Attributes for Jakarta EE Libraries

Attribute	Description
`Specification-Version`	An optional library specification version consisting of nonnegative integer components separated by periods, for example 2.0 or 9.0.1.1. Referencing descriptors can require a minimum version or, with exact-match=true, the supplied exact version. Deployment needs a matching registered library under those rules; an exact 2.0 provider is not required when a higher specification version is allowed. 2.0 and 2.0.0 compare as the same specification version. The specification version for a shared Jakarta EE library can also be set at the command-line when deploying the library, with some restrictions. See Deploying Shared Jakarta EE Libraries and Dependent Applications.
`Implementation-Version`	An implementation version requires a specification version when registering a library. Versions such as 1.9 and 1.10 are compared as integer components, so 1.10 is higher; they are not floating-point numbers. Resolution first selects the highest permitted specification version, or the supplied exact specification when exact-match=true. It then resolves an implementation at that selected specification. When all implementations there are numeric component versions, an omitted implementation selects the highest, and a non-exact numeric requirement selects the highest not below it. For a text or mixed implementation set, specify the required implementation explicitly; a text requirement matches that exact text. An omitted implementation can select a sole text implementation, but multiple unrankable implementations leave the reference unresolved. With exact-match=true, only the version fields actually supplied are pinned. Failure at the chosen specification does not cause fallback to a lower specification. The implementation version for a shared Jakarta EE library can also be set at the command-line when deploying the library, with some restrictions. See Deploying Shared Jakarta EE Libraries and Dependent Applications.

To specify attributes in a manifest file:

1. Open (or create) the manifest file using a text editor. For the example shared Jakarta EE library, you would use the commands:

   Any text editor may be used; emacs below is only an example.

   ```
   cd /apps/myLibrary
   mkdir -p META-INF
   emacs META-INF/MANIFEST.MF
   ```

   For the optional package example, use:

   ```
   cd /apps/myOptPkg
   mkdir -p META-INF
   emacs META-INF/MANIFEST.MF
   ```

2. In the text editor, add a string value to specify the name of the shared Jakarta EE library. For example:

   ```
   Extension-Name: myExtension
   ```

   Applications that reference the library must specify the exact `Extension-Name` in order to use the shared files.

3. As a best practice, enter the optional version information for the shared Jakarta EE library. For example:

   ```
   Extension-Name: myExtension
   Specification-Version: 2.0
   Implementation-Version: 9.0.0
   ```


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

Using the major/minor format for the version identifiers provides the most flexibility when referencing the library from another application (see Table 14-2)

> **Note**
>
> Although you can optionally specify the Specification-Version and Implementation-Version at the command line during deployment, Oracle recommends that you include these strings in the `MANIFEST.MF` file. Including version strings in the manifest ensures that you can deploy new versions of the library alongside older versions. See Deploying Shared Jakarta EE Libraries and Dependent Applications.

For the optional package assembled at /apps/myOptPkg, save the following complete manifest as /apps/myOptPkg/META-INF/MANIFEST.MF, including the final blank line.

```
Manifest-Version: 1.0
Extension-Name: myOptPkg
Specification-Version: 2.0
Implementation-Version: 9.0.0
```

Packaging Shared Jakarta EE Libraries for Distribution and Deployment

If you are delivering the shared Jakarta EE Library or optional package for deployment by an administrator, package the deployment files into an archive file (an `.EAR` file or standalone module archive file for shared Jakarta EE libraries, or a simple `.JAR` file for optional packages) for distribution. See Deploying Applications Using wldeploy.

Because a shared Jakarta EE library is packaged as a standard Jakarta EE application or standalone module, you may also choose to export a library's deployment configuration to a deployment plan, as described in *Deploying Applications to Oracle WebLogic Server*. Optional package `.JAR` files contain no deployment descriptors and cannot be exported.

For development purposes, you may choose to deploy libraries as exploded archive directories to facilitate repeated updates and redeployments.

Referencing Shared Jakarta EE Libraries in an Enterprise Application

A Jakarta EE application can reference a registered shared Jakarta EE library using entries in the application's `weblogic-application.xml` deployment descriptor.

Table 14-2 describes the XML elements that define a library reference.

Table 14-2 weblogic-application.xml Elements for Referencing a Shared Jakarta EE Library

Element	Description
`library-ref`	`library-ref` is the parent element in which you define a reference to a shared Jakarta EE library. Enclose all other elements within `library-ref`.

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

Table 14-2 (Cont.) weblogic-application.xml Elements for Referencing a Shared Jakarta EE Library

Element	Description
`library-name`	A required string value that specifies the name of the shared Jakarta EE library to use. `library-name` must exactly match the value of the `Extension-Name` attribute in the library's manifest file. (See Table 14-1.)
`specification-version`	An optional String value that defines the required specification version of the shared Jakarta EE library. If this element is not set, the application uses a matching library with the highest specification version. If you specify a string value using major/minor version format, the application uses a matching library with the highest specification version that is not below the configured value. If all available libraries are below the configured `specification-version`, the application cannot be deployed. The required version can be further constrained by using the `exact-match` element, described below. Specification versions must use the numeric component format; 9.2BETA is invalid as a specification version. 2.0 and 2.0.0 compare as the same specification version.
`implementation-version`	Versions such as 1.9 and 1.10 are compared as integer components, so 1.10 is higher; they are not floating-point numbers. Resolution first selects the highest permitted specification version, or the supplied exact specification when exact-match=true. It then resolves an implementation at that selected specification. When all implementations there are numeric component versions, an omitted implementation selects the highest, and a non-exact numeric requirement selects the highest not below it. For a text or mixed implementation set, specify the required implementation explicitly; a text requirement matches that exact text. An omitted implementation can select a sole text implementation, but multiple unrankable implementations leave the reference unresolved. With exact-match=true, only the version fields actually supplied are pinned. Failure at the chosen specification does not cause fallback to a lower specification.
`exact-match`	An optional Boolean value that determines whether the application should use a shared Jakarta EE library with a higher specification or implementation version than the configured value, if one is available. By default this element is false, which means that WebLogic Server uses higher-versioned libraries if they are available. Set this element to true to require the exact matching version as specified in the `specification-version` and `implementation-version` elements.
`context-root`	An optional alternate context root for a WAR library referenced by an EAR. An EAR can reference a WAR library and use this element to supply that WAR module's context root. When a WAR references another WAR through weblogic.xml, this child is ignored.

For example, this simple entry in the `weblogic-application.xml` descriptor references a shared Jakarta EE library, `myLibrary`:

```
<library-ref>
   <library-name>myLibrary</library-name>
</library-ref>
```

In the above example, WebLogic Server attempts to find a library named myLibrary when deploying the dependent application. If more than one copy of myLibrary is registered, WebLogic Server selects the library with the highest specification version. If multiple copies of the library use the selected specification version, WebLogic Server selects the copy having the highest implementation version.

This example references a shared Jakarta EE library with a requirement for the specification version:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
</library-ref>
```

Versions such as 1.9 and 1.10 are compared as integer components, so 1.10 is higher; they are not floating-point numbers. Resolution first selects the highest permitted specification

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

version, or the supplied exact specification when exact-match=true. It then resolves an implementation at that selected specification. When all implementations there are numeric component versions, an omitted implementation selects the highest, and a non-exact numeric requirement selects the highest not below it. For a text or mixed implementation set, specify the required implementation explicitly; a text requirement matches that exact text. An omitted implementation can select a sole text implementation, but multiple unrankable implementations leave the reference unresolved. With exact-match=true, only the version fields actually supplied are pinned. Failure at the chosen specification does not cause fallback to a lower specification.

This example references a shared Jakarta EE library with both a specification version and a text implementation version:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
   <implementation-version>81Beta</implementation-version>
</library-ref>
```

The selected specification must contain implementation 81Beta; use exact-match=true with both versions if a specific known pair is required.

The following example requires an exact match for both the specification and implementation versions:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
   <implementation-version>8.1</implementation-version>
   <exact-match>true</exact-match>
</library-ref>
```

The following example specifies a `context-root` with the library reference. When a WAR library reference is made from `weblogic-application.xml`, the `context-root` may be specified with the reference:

```
<library-ref>
   <library-name>myLibrary</library-name>
   <context-root>mywebapp</context-root>
</library-ref>
```

Overriding context-roots Within a Referenced Enterprise Library

A Jakarta EE application can override `context-roots` within a referenced EAR library using entries in the application's `weblogic-application.xml` deployment descriptor. Table 14-3 describes the XML elements that override `context-root` in a library reference.

Table 14-3 weblogic-application.xml Elements for Overriding a Shared Jakarta EE Library

Element	Description
`context-root`	An optional String value that overrides the `context-root` elements declared in libraries. In the absence of this element, the library's `context-root` is used. Only a referencing application (for example, a user application) can override the `context-root` elements declared in its libraries.
`override-value`	An optional String value that specifies the value of the `library-context-root-override` element when overriding the context-root elements declared in libraries. In the absence of these elements, the library's `context-root` is used.

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

The following example specifies a `context-root-override`, which in turn, refers to the old `context-root` specified in one of its libraries and the new `context-root` that should be used instead. (override):

```
<library-ref>
   <library-name>myLibrary</library-name>
   <specification-version>2.0</specification-version>
   <implementation-version>8.1</implementation-version>
   <exact-match>true</exact-match>
</library-ref>
<library-context-root-override>
   <context-root>webapp</context-root>
   <override-value>mywebapp</override-value>
</library-context-root-override>
```

In the above example, the current application refers to `myLibrary`, which contains a Web application with a `context-root` of `webapp`. The only way to override this reference is to declare a `library-context-root-override` that maps `webapp` to `mywebapp`.

URIs for Shared Jakarta EE Libraries Deployed As a Standalone Module

When referencing the URI of a shared Jakarta EE library that was deployed as a standalone module (EJB or Web application), note that the module URI corresponds to the deployment name of the shared Jakarta EE library. This can be a name that was manually assigned during deployment, the name of the archive file that was deployed, or the name of the exploded archive directory that was deployed. If you redeploy the same module using a different file name or from a different location, the default deployment name also changes and referencing applications must be updated to use the correct URI.

To avoid this problem, deploy all shared Jakarta EE libraries as enterprise applications, rather than as standalone modules. If you choose to deploy a library as a standalone Jakarta EE module, always specify a known deployment name and use that name as the URI in referencing applications.

Referencing Optional Packages from a Jakarta EE Application or Module

Any Jakarta EE archive (JAR, WAR, RAR, EAR) can reference one or more registered optional packages using attributes in the archive's manifest file.

Table 14-4 Manifest Attributes for Referencing Optional Packages

Attribute	Description
`Extension-List` *`logical_name`* [...]	Extension-List contains a space-separated list of dependency tokens. A required String value that defines a logical name for an optional package dependency. You can use multiple values in the `Extension-List` attribute to designate multiple optional package dependencies. For example: `Extension-List: dependency1 dependency2`
logical_name`-Extension-Name`	A required string value that identifies the name of an optional package dependency. This value must match the `Extension-Name` attribute defined in the optional package's manifest file. For every dependency, including a single dependency, prepend its token from Extension-List followed by a hyphen. `dependency1-Extension-Name: myOptPkg`

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

Table 14-4 (Cont.) Manifest Attributes for Referencing Optional Packages

Attribute	Description
logical_name`-Specification-Version`	An optional numeric specification-version requirement. Select the highest specification version not below the requirement; 2.0 and 2.0.0 compare as the same version. Specification versions must use the numeric component format; 9.2BETA is invalid as a specification version. Invalid optional-package specification metadata may be warned about and ignored by the manifest parser. The optional-package reference is unresolved when no permitted provider exists; the deployment outcome follows the policy below. For every dependency, including a single dependency, prepend its token from Extension-List followed by a hyphen.
logical_name`-Implementation-Version`	Versions such as 1.9 and 1.10 are compared as integer components, so 1.10 is higher; they are not floating-point numbers. Resolution first selects the highest permitted specification version, or the supplied exact specification when exact-match=true. It then resolves an implementation at that selected specification. When all implementations there are numeric component versions, an omitted implementation selects the highest, and a non-exact numeric requirement selects the highest not below it. For a text or mixed implementation set, specify the required implementation explicitly; a text requirement matches that exact text. An omitted implementation can select a sole text implementation, but multiple unrankable implementations leave the reference unresolved. With exact-match=true, only the version fields actually supplied are pinned. Failure at the chosen specification does not cause fallback to a lower specification. The optional-package reference is unresolved when these rules cannot be satisfied; the deployment outcome follows the policy below. For every dependency, including a single dependency, prepend its token from Extension-List followed by a hyphen.

```
Manifest-Version: 1.0
Extension-List: dependency1
dependency1-Extension-Name: myOptPkg
dependency1-Specification-Version: 2.0
```

For example, this simple entry in the manifest file for a dependent archive references two optional packages, `myAppPkg` and `my3rdPartyPkg`:

```
Extension-List: internal 3rdparty
internal-Extension-Name: myAppPkg
3rdparty-Extension-Name: my3rdPartyPkg
```

This example requires a specification version of 2.0 or higher for `myAppPkg`:

```
Extension-List: internal 3rdparty
internal-Extension-Name: myAppPkg
3rdparty-Extension-Name: my3rdPartyPkg
internal-Specification-Version: 2.0
```

This example requires a specification version of 2.0 or higher for `myAppPkg`, and an exact match for the implementation version of `my3rdPartyPkg`:

```
Extension-List: internal 3rdparty
internal-Extension-Name: myAppPkg
3rdparty-Extension-Name: my3rdPartyPkg
internal-Specification-Version: 2.0
3rdparty-Implementation-Version: 8.1GA
```


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

By default, unresolved registered optional-package references produce a warning and deployment continues. To make unresolved optional-package references fail deployment, set -Dweblogic.application.RequireOptionalPackages=true on each relevant server JVM at startup. This switch controls unresolved optional-package references; it is not a library deployment command or a validator for malformed manifest attributes.

Using weblogic.appmerge to Merge Libraries

You can use `weblogic.appmerge` to understand a library merge by examining the merged application you have written to disk.

Use weblogic.appmerge to inspect an application with its referenced libraries merged into the output. It merges application contents and descriptors and can write the result to an archive or directory. Run it after setting the WebLogic Server development environment and preparing the application and library files.

Using weblogic.appmerge from the CLI

Invoke `weblogic.appmerge` using the following syntax:

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

where valid options are shown in Table 14-5:

Table 14-5 weblogic.appmerge Options

Option	Comment
`-help`	Print the standard usage message.
`-version`	Print version information.
`-output <file>`	Specifies an alternate output archive or directory. If not set, output is placed in the source archive or directory.
`-plan <file>`	Specifies an optional deployment plan.
`-verbose`	Provide more verbose output.
`-library <file>`	Comma-separated list of libraries. Each library may optionally set its name and versions, if not already set in its manifest, using the following syntax: `<file> [@name=<string>@libspecver=<version> @libimplver=<version|string>].`
`-librarydir <dir>`	Registers all files in specified directory as libraries.
`-writeInferredDescriptors`	Writes deployment descriptors containing information inferred from annotations.
-altappdd <file>	Uses an alternate descriptor file; supported for both application and single-module inputs.

Example:

Run this example in a work directory containing SportsApp.ear, Weather.war and Calendar.ear. The library names and versions must satisfy the references in SportsApp.ear. CompleteSportsApp.ear is the merged output. Choose an output path distinct from your input to retain the original application; without -output the tool writes to the source location.

```
java weblogic.appmerge \
  -output CompleteSportsApp.ear \
```


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

```
  -library Weather.war,Calendar.ear \
  SportsApp.ear
```

Using weblogic.appmerge as an Ant Task

The ant task provides similar functionality as the command line utility. It supports `source`, `output`, `libraryDir`, `plan` and `verbose` attributes as well as multiple `<library>` sub-elements. Here is an example:

```
<taskdef name="appmerge" classname="weblogic.ant.taskdefs.j2ee.AppMergeTask"/>
<appmerge source="SportsApp.ear" output="CompleteSportsApp.ear">
   <library file="Weather.war"/>
   <library file="Calendar.ear"/>
</appmerge>
```

For archive-file inputs and outputs, this task can skip the merge when the output file is newer than the source file. This shortcut does not check whether referenced libraries or the deployment plan have changed. When those inputs change, remove the generated merged output before rerunning the task, or use a clean target that removes that generated output. Keep the original application and libraries separate from the generated output.

Integrating Shared Jakarta EE Libraries with the Split Development Directory Environment

You can generate a basic *build.xml* file in the shared Jakarta EE library directories and then build the applications in a split development directory.

The `BuildXMLGen` includes a `-librarydir` option to generate build targets that include one or more shared Jakarta EE library directories. See Generating a Basic build.xml File Using weblogic.BuildXMLGen.

The `wlcompile` and `wlappc` Ant tasks include a `librarydir` attribute and `library` element to specify one or more shared Jakarta EE library directories to include in the classpath for application builds. See Building Applications in a Split Development Directory.

The librarydir/library inputs make library contents available to the build tools. They do not deploy or register those libraries on a running server. Before deploying the dependent application, register its required libraries on every target server as described in Deploying Shared Jakarta EE Libraries and Dependent Applications.Deploying Shared Jakarta EE Libraries and Dependent Applications

Deploying Shared Jakarta EE Libraries and Dependent Applications

Shared Jakarta EE libraries are registered with one or more WebLogic Server instances by deploying them to the target servers and indicating that the deployments are to be shared. Shared Jakarta EE libraries must be targeted to the same WebLogic Server instances you want to deploy applications that reference the libraries.

If you try to deploy a referencing application to a server instance that has not registered a required library, deployment of the referencing application fails. See Registering Libraries with WebLogic Server in *Deploying Applications to Oracle WebLogic Server* for more information.

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

See Deploying Applications the *Oracle WebLogic Remote Console Online Help* for detailed instructions on installing (deploying) a shared Jakarta EE library using the WebLogic Remote Console and for targeting the library to the server or cluster to which the application that is referencing the library is also targeted.

If you use the `wldeploy` Ant task as part of your iterative development process, use the `library`, `libImplVer`, and `libSpecVer` attributes to deploy a shared Jakarta EE library. See wldeploy Ant Task Reference, for details and examples.

Required shared-library references must resolve on every target server to a registered library meeting the reference's version rules. Manifest optional-package references follow the warning/failure policy described in Referencing Optional Packages from a Jakarta EE Application or Module.Referencing Optional Packages from a Jakarta EE Application or Module

Web Application Shared Jakarta EE Library Information

This section describes library references from a Web application's weblogic.xml. A Web application can reference registered WAR libraries and plain-JAR libraries. A WAR library can also be referenced by an EAR through weblogic-application.xml.

> **Note**
>
> This section describes library references from a Web application's weblogic.xml. A Web application can reference registered WAR libraries and plain-JAR libraries. A WAR library can also be referenced by an EAR through weblogic-application.xml.

In particular:

- WAR libraries supply Web resources and classes; plain-JAR libraries supply classes.
- Rather than update the `weblogic-application.xml` file, Web applications reference Web application libraries by updating the `weblogic.xml` deployment descriptor file. The elements are almost the same as those described in Referencing Shared Jakarta EE Libraries in an Enterprise Application; the only difference is that the `<context-root>` child element of `<library-ref>` is ignored in this case.
- A Web application cannot use weblogic.xml to reference an EJB module library or an EAR library.

Other than these differences in how they are referenced, the way to create, package, and deploy a Web application library is the same as that of a standard shared Jakarta EE library.

Using WebApp Libraries With Web Applications

Just as standard shared Jakarta EE applications can be deployed to WebLogic Server as `application-libraries`, a standard Web application can be deployed to WebLogic Server as a `webapp-library` so that other Web applications can refer to these libraries.

Web application libraries facilitate the reuse of code and resources. Such libraries also help you separate out third-party Web applications or frameworks that your Web application might be using. Furthermore, common resources can be packaged separately as libraries and referenced in different Web applications, so that you don't have to bundle them with each Web application. When you include a `webapp-library` in your Web application, at deployment time the container merges all the static resources, classes, and `JAR` files into your Web application.

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

The first step in using a WebApp library is to register a Web application as a `webapp-library`. This can be accomplished by deploying a Web application using either the WebLogic Remote Console or the `weblogic.Deployer` tool as a library. To make other Web applications refer to this library, their `weblogic.xml` file must have a `library-ref` element pointing to the `webapp-library`, as follows:

```
<library-ref>
<library-name>BaseWebApp</library-name>
<specification-version>2.0</specification-version>
<implementation-version>8.1beta</implementation-version>
<exact-match>false</exact-match>
</library-ref>
```

8.1beta is a text implementation identifier and must match exactly even though exact-match is false; specification selection still follows the permitted specification rule.

When multiple libraries are present, the CLASSPATH/resource path precedence order follows the order in which the library-ref elements appear in the weblogic.xml file.

Accessing Registered Shared Jakarta EE Library Information with LibraryRuntimeMBean

You can use different types of MBeans to obtain information about the shared Jakarta EE library and access the libraries that the applications use.

Each deployed shared Jakarta EE library is represented by a `LibraryRuntimeMBean`. You can use this MBean to obtain information about the library itself, such as its name or version. You can also obtain the `ApplicationRuntimeMBeans` associated with deployed applications. `ApplicationRuntimeMBean` provides two methods to access the libraries that the application is using:

- `getLibraryRuntimes()` returns the shared Jakarta EE libraries referenced in the `weblogic-application.xml` file.
- `getOptionalPackageRuntimes()` returns the optional packages referenced in the manifest file.

See the *Java API Reference for Oracle WebLogic Server*.

Order of Precedence of Modules When Referencing Shared Jakarta EE Libraries

When an enterprise application references one or more shared Jakarta EE libraries, and the application is deployed to WebLogic Server, the server internally merges the information in the `weblogic-application.xml` file of the referencing enterprise application with the information in the deployment descriptors of the referenced libraries.

The order in which WebLogic Server internally merges the information is as follows:

1. When the enterprise application is deployed, WebLogic Server reads its `weblogic-application.xml` deployment descriptor.
2. WebLogic Server reads the deployment descriptors of any referenced shared Jakarta EE libraries. Depending on the type of library (enterprise application, EJB, or Web application), the read file might be `weblogic-application.xml`, `weblogic.xml`, `weblogic-ejb-jar.xml`, and so on.

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

3. WebLogic Server first merges the referenced shared Jakarta EE library deployment descriptors (in the order in which they are referenced, one at a time) and then merges the `weblogic-application.xml` file of the referencing enterprise application on top of the library descriptor files.

As a result of the way the descriptor files are merged, the elements in the descriptors of the shared Jakarta EE libraries referenced first in the `weblogic-application.xml` file have precedence over the ones listed last. The elements of the enterprise application's descriptor itself have precedence over all elements in the library descriptors.

For example, assume that an enterprise application called `myApp` references two shared Jakarta EE libraries (themselves packaged as enterprise applications): `myLibA` and `myLibB`, in that order. Both the `myApp` and `myLibA` applications include an EJB module called `myEJB`, and both the `myLibA` and `myLibB` applications include an EJB module called `myOtherEJB`.

Further assume that once the `myApp` application is deployed, a client invokes, via the `myApp` application, the `myEJB` module. In this case, WebLogic Server actually invokes the EJB in the `myApp` application (rather than the one in `myLibA`) because modules in the *referencing* application have higher precedence over modules in the *referenced* applications. If a client invokes the `myOtherEJB` EJB, then WebLogic Server invokes the one in `myLibA`, because the library is referenced first in the `weblogic-application.xml` file of `myApp`, and thus has precedence over the EJB with the same name in the `myLibB` application.

Best Practices for Using Shared Jakarta EE Libraries

Keep in mind these best practices when developing shared Jakarta EE libraries and optional packages:

- Use shared Jakarta EE Libraries when you want to share one or more Jakarta EE modules (EJBs, Web applications, enterprise applications, or plain Java classes) with multiple enterprise applications.
- If you need to deploy a standalone Jakarta EE module, such as an EJB JAR file, as a shared Jakarta EE library, package the module within an enterprise application. Doing so avoids potential URI conflicts, because the library URI of a standalone module is derived from the deployment name.
- If you choose to deploy a shared Jakarta EE library as a standalone Jakarta EE module, always specify a known deployment name during deployment and use that name as the URI in referencing applications.
- Use optional packages when multiple Jakarta EE archive files need to share a set of Java classes.
- If you have a set of classes that must be available to applications in an entire domain, and you do not frequently update those classes (for example, if you need to share 3rd party classes in a domain), use the domain /lib subdirectory rather than using shared Jakarta EE libraries or optional packages. Classes in the /lib subdirectory are made available (through the separate domain-lib classloader) to all Jakarta EE applications running on WebLogic Server instances in the domain.
- Always specify a specification version and implementation version, even if you do not intend to enforce version requirements with dependent applications. Specifying versions for shared Jakarta EE libraries enables you to deploy multiple versions of the shared files for testing.
- Always specify an `Extension-Name` value for each shared Jakarta EE library. If you do not specify an extension name, one is derived from the deployment name of the library. Default

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

deployment names are different for archive and exploded archive deployments, and they can be set to arbitrary values in the deployment command.

- When developing a Web application for deployment as a shared Jakarta EE library, use a unique context root. If the context root conflicts with the context root in a dependent Jakarta EE application, use the `context-root` element in the EAR's `weblogic-application.xml` deployment descriptor to override the library's context root.
- Package shared Jakarta EE libraries as archive files for delivery to administrators or deployers in your organization. Deploy libraries from exploded archive directories during development to allow for easy updates and repeated redeployments.
- Deploy shared libraries to the same targets as their consumers. Required shared-library references must resolve on every target server to a registered library meeting the reference’s version rules. Manifest optional-package references follow the warning/failure policy described in Referencing Optional Packages from a Jakarta EE Application or Module.Referencing Optional Packages from a Jakarta EE Application or Module

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

15

Programming Application Life Cycle Events

Learn how to create applications that respond to WebLogic Server application life cycle events. This chapter includes the following sections:

Understanding Application Life Cycle Events

Application life cycle listener events provide handles on which developers can control behavior during deployment, undeployment, and redeployment. Learn how you can use the application life cycle listener events.

Four application life cycle events are provided with WebLogic Server, which can be used to extend listener, shutdown, and startup classes. These include:

- Listeners—attachable to any event. Possible methods for Listeners are:
  - `public void preStart(ApplicationLifecycleEvent evt) {}`

    preStart is invoked during application prepare, before preparation of the application has completed.
  - `public void postStart(ApplicationLifecycleEvent evt) {}`

    postStart is invoked near the end of application activation, after application initialization.
  - `public void preStop(ApplicationLifecycleEvent evt) {}`

    preStop is invoked as application deactivation begins.
  - `public void postStop(ApplicationLifecycleEvent evt) {}`

    postStop is invoked during application unprepare, after the application has been deactivated. It is not a callback for the separate remove phase.
- Shutdown classes only get postStop events.

  > **Note**
  >
  > Application-scoped shutdown classes have been deprecated as of release 9.0 of WebLogic Server. Use life cycle listeners instead.

- Startup classes only get preStart events.

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

> **Note**
>
> Application-scoped startup classes have been deprecated since WebLogic Server 9.0. Use application life cycle listeners instead.
>
> For the legacy startup and shutdown elements, provide public static void main(String[] args). WebLogic invokes that method with an empty argument array. A startup class is invoked only at preStart; a shutdown class is invoked only at postStop. Declaring listener-named methods on those classes does not register them as listener callbacks.
>
> ApplicationLifecycleListener has no remove callback. Use preStop/postStop for application shutdown work; these callbacks follow the application lifecycle transitions described above. Application-version deletion notifications use ApplicationVersionLifecycleListener.postDelete, described later in this chapter.

Registering Events in weblogic-application.xml

You must register the application life cycle listener classes in the `weblogic-application.xml` deployment descriptor in order to use them.

See Enterprise Application Deployment Descriptor Elements. Define the following elements:

- `listener`—Used to register user defined application life cycle listeners. These are classes that extend the abstract base class `weblogic.application.ApplicationLifecycleListener`.
- `shutdown`—Used to register user-defined shutdown classes.
- `startup`—Used to register user-defined startup classes.

Programming Basic Life Cycle Listener Functionality

You can create a listener by extending the abstract class (provided with WebLogic Server) `weblogic.application.ApplicationLifecycleListener`. The container then searches for your listener.

You override the following methods provided in the WebLogic Server `ApplicationLifecycleListener` abstract class to extend your application and add any required functionality:

- `preStart{}`
- `postStart{}`
- `preStop{}`
- `postStop{}`

Example 15-1 illustrates how you override the ApplicationLifecycleListener. In this example, the public class MyListener extends ApplicationLifecycleListener.

Example 15-1 MyListener

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyListener extends ApplicationLifecycleListener {
  public void preStart(ApplicationLifecycleEvent evt) {
     System.out.println
```


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

```
    ("MyListener(preStart) -- we should always see you..");
   } // preStart
  public void postStart(ApplicationLifecycleEvent evt) {
     System.out.println
     ("MyListener(postStart) -- we should always see you..");
   } // postStart
  public void preStop(ApplicationLifecycleEvent evt) {
     System.out.println
     ("MyListener(preStop) -- we should always see you..");
   } // preStop
  public void postStop(ApplicationLifecycleEvent evt) {
     System.out.println
     ("MyListener(postStop) -- we should always see you..");
   } // postStop
   public static void main(String[] args) {
     System.out.println
     ("MyListener(main): in main .. we should never see you..");
   } // main
}
```

Example 15-2 illustrates how you implement the shutdown class. The shutdown class is attachable to preStop and postStop events. In this example, the public class `MyShutdown` does not extend `ApplicationLifecycleListener` because a shutdown class declared in the `weblogic-application.xml` deployment descriptor does not need to depend on any WebLogic Server-specific interfaces.

Example 15-2 MyShutdown

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyShutdown {
   public static void main(String[] args) {
     System.out.println
     ("MyShutdown(main): in main .. should be for post-stop");
   } // main
}
```

Example 15-3 illustrates how you implement the startup class. The startup class is attachable to preStart and postStart events. In this example, the public class `MyStartup` does not extend `ApplicationLifecycleListener` because a startup class declared in the `weblogic-application.xml` deployment descriptor does not need to depend on any WebLogic Server-specific interfaces.

Example 15-3 MyStartup

```
import weblogic.application.ApplicationLifecycleListener;
import weblogic.application.ApplicationLifecycleEvent;
public class MyStartup {
   public static void main(String[] args) {
     System.out.println
     ("MyStartup(main): in main .. should be for pre-start");
   } // main
}
```

Configuring a Role-Based Application Life Cycle Listener

For an ApplicationLifecycleListener registered with listener, run-as-principal-name selects the user identity under which its callbacks execute. However, if the run-as-principal-name identity defined for the application life cycle listener is an administrator, the application deployer must have administrator privileges; otherwise, deployment will fail.

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

1. Follow the basic programming steps outlined in Programming Basic Life Cycle Listener Functionality.
2. Within listener, add run-as-principal-name to select the callback execution identity. This does not grant a user permission to deploy or start/stop the application.

```
<listener>
  <listener-class>MyListener</listener-class>
  <run-as-principal-name>javajoe</run-as-principal-name>
</listener>
```

   javajoe is an illustrative name; use an existing user in the application's security realm.

The identity specified here should be a valid user name in the system. If `run-as-principal-name` is not specified, the deployment initiator user identity will be used as the `run-as` identity for the execution of the application life cycle listener.

This identity setting applies to ApplicationLifecycleListener callbacks; application-version callbacks use the anonymous identity described in Understanding Application Version Life Cycle Event Behavior.

Examples of Configuring Life Cycle Events with and without the URI Parameter

You can configure application life cycle events with or without specifying URI elements in the `weblogic-application.xml` deployment descriptor file.

The optional listener-uri, startup-uri and shutdown-uri elements identify the JAR containing the corresponding class, relative to the EAR root. If a URI element is omitted, that class must already be visible to the application classloader. For the examples here, place compiled classes with their package paths under APP-INF/classes, or package them in a JAR under APP-INF/lib. Classes visible only to a child module classloader, such as WEB-INF/classes in a WAR, are not sufficient. No shell CLASSPATH edit is required for these EAR layouts.

The startup and shutdown portions of these descriptor examples are compatibility examples for deprecated application-scoped classes.

The descriptor elements listener-uri, startup-uri and shutdown-uri each name foo.jar, located at the EAR root (myEar/foo.jar). They are descriptor elements, not command-line parameters.

Example 15-4 Configuring Application Life Cycle Events Using the URI Parameter

```
<listener>
     <listener-class>MyListener</listener-class>
     <listener-uri>foo.jar</listener-uri>
</listener>
<startup>
     <startup-class>MyStartup</startup-class>
     <startup-uri>foo.jar</startup-uri>
</startup>
<shutdown>
     <shutdown-class>MyShutdown</shutdown-class>
     <shutdown-uri>foo.jar</shutdown-uri>
</shutdown>
```

The following example illustrates how you configure application life cycle events without specifying URI elements.

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

Example 15-5 Configuring Application Life Cycle Events without Using the URI Parameter

```
<listener>
      <listener-class>MyListener</listener-class>
 </listener>
 <startup>
      <startup-class>MyStartup</startup-class>
 </startup>
 <shutdown>
      <shutdown-class>MyShutdown</shutdown-class>
 </shutdown>
```

The following minimal complete descriptor registers MyListener alone. Add its listener child to an existing descriptor while retaining other configuration. For this default-package teaching example, put MyListener.class in APP-INF/classes/MyListener.class. If you use a package, preserve its directory layout and use its fully qualified name in listener-class.

```
<weblogic-application xmlns="http://xmlns.oracle.com/weblogic/weblogic-
application">
  <listener>
    <listener-class>MyListener</listener-class>
  </listener>
</weblogic-application>
```

Understanding Application Life Cycle Event Behavior During Redeployment

Application life cycle events are only triggered if a full redeployment of the application occurs. During a full redeployment of the application—provided the application life cycle events have been registered—the application life cycle first commences the shutdown sequence, next re-initializes its classes, and then performs the startup sequence.

Application life cycle callbacks run during initial deployment and subsequent whole-application lifecycle transitions, including undeployment and full redeployment. A partial module/resource update does not by itself run this whole-application callback sequence. For a normal successful in-place full redeployment, the old application receives preStop(evt), then postStop(evt); the replacement application is prepared and receives preStart(evt), then postStart(evt) during activation. Initialization of application and listener classes occurs as required during preparation; it is not a separate listener callback. During production redeployment with distinct application versions, the new version can start while the old version is still serving existing work, so do not assume that all old-version stop callbacks precede new-version start callbacks. See Developing Applications for Production Redeployment and Programming Application Version Life Cycle Events.Developing Applications for Production RedeploymentProgramming Application Version Life Cycle Events

Programming Application Version Life Cycle Events

Learn how to create applications that respond to WebLogic Server application version life cycle events.

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

Understanding Application Version Life Cycle Event Behavior

WebLogic Server provides application version life cycle event notifications by allowing you to extend the `ApplicationVersionLifecycleListener` class and specify a life cycle listener in `weblogic-application.xml`. See Enterprise Application Deployment Descriptor Elements and Examples of Configuring Life Cycle Events with and without the URI Parameter.

Application version life cycle events are invoked:

- For both static and dynamic deployments.
- ApplicationVersionLifecycleListener callbacks execute as the anonymous identity; run-as-principal-name does not change their callback identity.
- Only if the current application is versioned; otherwise, version life cycle events are ignored.
- A registered version listener can observe events for other versions of the same application and for its own version when it is registered at the time of notification. Newly deployed listeners are registered during prepare, after the initial preDeploy notification. For a normal-mode postDeploy notification, recipients are the newly deployed version and other versions that were active when the notification was dispatched. Retired or unregistered listeners are not guaranteed every later event. Use evt.isOwnVersion() to distinguish an event for the listener's own version from an event for another version; evt.getApplicationId() identifies the version whose operation caused the event.

Types of Application Version Life Cycle Events

Four application version life cycle events are provided with WebLogic Server:

- `public void preDeploy(ApplicationVersionLifecycleEvent evt)`
  - The `preDeploy` event is invoked when an application version deploy or redeploy operation is initiated.
- `public void postDeploy(ApplicationVersionLifecycleEvent evt)`
  - The `postDeploy` event is invoked when an application version is deployed or redeployed successfully.
- `public void preUndeploy(ApplicationVersionLifecycleEvent evt)`
  - The `preUndeploy` event is invoked when an application version undeploy operation is initiated.
- `public void postDelete(ApplicationVersionLifecycleEvent evt)`
  - The `postDelete` event is invoked when an application version is deleted.

> **Note**
>
> A `postDelete` event is only fired after the entire application version is completely removed. It does not include a partial undeploy, such as undeploying a module or from a subset of targets.

Compile the following default-package class to APP-INF/classes/ MyVersionListener.class so it is visible to the EAR application loader. For a packaged class, use its fully qualified class name. Put the registration fragment inside META-INF/

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

weblogic-application.xml. The EAR must carry the Chapter 7 application version identifier.

preDeploy and preUndeploy may veto their operation by throwing ApplicationException. If an operation is vetoed or has already failed, later listeners may not be called. postDeploy and postDelete are notifications, not veto points. This logging example does not veto anything.

```
import weblogic.application.ApplicationException;
import weblogic.application.ApplicationVersionLifecycleEvent;
import weblogic.application.ApplicationVersionLifecycleListener;

public class MyVersionListener extends
ApplicationVersionLifecycleListener {
    @Override
    public void preDeploy(ApplicationVersionLifecycleEvent evt)
            throws ApplicationException {
        report("preDeploy", evt);
    }

    @Override
    public void postDeploy(ApplicationVersionLifecycleEvent evt) {
        report("postDeploy", evt);
    }

    @Override
    public void preUndeploy(ApplicationVersionLifecycleEvent evt)
            throws ApplicationException {
        report("preUndeploy", evt);
    }

    @Override
    public void postDelete(ApplicationVersionLifecycleEvent evt) {
        report("postDelete", evt);
    }

    private static void report(String phase,
ApplicationVersionLifecycleEvent evt) {
        System.out.println(phase + ": " + evt.getApplicationId()
                + ", ownVersion=" + evt.isOwnVersion());
    }
}


<listener>
  <listener-class>MyVersionListener</listener-class>
</listener>
```

Example of Production Deployment Sequence When Using Application Version Life Cycle Events

This example assumes normal-mode deployment of V1 followed by production deployment of V2, retirement of V1, and undeployment of V2. It distinguishes dispatch of an application-version event from receipt by an already registered ApplicationVersionLifecycleListener. It is a conceptual successful-path sequence, not a global clock across target servers.

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

Table 15-1 Sequence of Deployment Actions and Application Version Life Cycle Events

Normal successful operation	Event version and notification point	Registered observers
Initial deployment of V1	preDeploy(V1) before V1 is prepared	Existing registered versions of this application, if any; V1 has no listener registered yet
V1 preparation and activation	V1 listeners register during prepare; postDeploy(V1) is dispatched when activation reaches the version-notification flow	V1 receives its own postDeploy event
Production deployment of V2 while V1 is active	preDeploy(V2) before V2 is prepared	V1 receives an event for V2; V2 has no listener registered yet
V2 preparation and activation	V2 listeners register during prepare; normal-mode postDeploy(V2) is dispatched as V2 becomes active and V1 loses active status	Both newly registered V2 and previously active V1 receive an event for V2
Retirement of V1	V1 finishes serving its existing work and its listeners are unregistered during unprepare	Do not represent this as an additional postDeploy callback
Undeployment of V2 begins	preUndeploy(V2) is dispatched at the undeployability check	V2 and any other still-registered versions of the same application can observe the event and veto through preUndeploy
V2 is unprepared and removed	V2 listeners are cleared during unprepare; postDelete(V2) is dispatched at the remove notification point	Only observers still registered at that point can receive it; V2 cannot rely on its already-cleared listener receiving its own postDelete

Post notifications are not unconditional failure/completion callbacks. A failure before the corresponding notification point can prevent delivery; reaching a callback does not certify that every later deployment-management step succeeded. preDeploy/preUndeploy can be vetoed. postDelete describes complete application-version removal, not a partial module or target undeploy. Ordinary ApplicationLifecycleListener callbacks such as postStart are separate from these version callbacks.

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

16

Programming Context Propagation

Learn how to use the context propagation APIs in WebLogic Server applications. This chapter includes the following sections:

Understanding Context Propagation

Context propagation allows programmers to associate information with an application, which is then carried along with every request. Furthermore, downstream components can add or modify this information so that it can be carried back to the originator.

WebLogic context propagation associates named WorkContext entries with the current thread through a WorkContextMap. For supported invocation paths, entries travel with a request according to their propagation modes and applicable propagation policies. Supported request/response paths can return updated entries to the caller.

Common use-cases for context propagation are any type of application in which information, usually related to the request, needs to be carried outside the application or to another application, rather than the information being an integral part of the application. Examples of these use cases include diagnostics monitoring, application transactions, and application load-balancing. The ability of context propagation to tie information to a request greatly simplifies managing such data, in contrast to maintaining a map of request data in each application and then implementing custom code to transmit such information between applications or threads. The mechanism is also called work areas or work contexts. It carries contextual metadata; it does not by itself begin, commit or roll back a Jakarta Transaction.

However, context propagation can occur within an application. For example, if an application submits work through a Work Manager, part of the processing occurs in different threads. Context propagation uses a `PropagationMode` to carry information to other threads.

Include PropagationMode.WORK when the entry must be copied into supported Work execution.

Programming context propagation has two parts: obtain the current WorkContextMap through the documented JNDI lookup and create WorkContext values in the client, then retrieve and use the entries in the invoked application. Supported uses include EJBs, servlets, Web services and JMS topics or queues. See Programming Context Propagation: Main Steps for details.

The WebLogic context propagation APIs are in the `weblogic.workarea` package. The following table describes the main interfaces and classes.

Table 16-1 Interfaces and classes of the WebLogic Context Propagation API

Interface or Class	Description
`WorkContextMap` Interface	Main context propagation interface used to tag applications with data and propagate that information via application requests. WorkContextMap is part of the client or application's JNDI environment and can be accessed through JNDI by looking up the name java:comp/WorkContextMap.

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

Table 16-1 (Cont.) Interfaces and classes of the WebLogic Context Propagation API

Interface or Class	Description
`WorkContext` Interface	WorkContext defines readContext and writeContext for marshaling and unmarshaling context values. Built-in value implementations include AsciiWorkContext, LongWorkContext, StringWorkContext and SerializableWorkContext. WorkContext itself does not declare get(). Its PrimitiveWorkContext subinterface exposes the contained value through Object get(); check the context and contained-value types before casting.
`WorkContextOutput/Input` Interfaces	Interfaces representing primitive streams used for marshaling and unmarshaling, respectively, `WorkContext` implementations.
`PropagationMode` Interface	Defines the propagation properties of WorkContext entries. The put(key, value) overload uses DEFAULT, which equals GLOBAL = RMI | JMS_QUEUE | SOAP | MIME_HEADER in this release. It does not include WORK or JMS_TOPIC. Combine required flags with bitwise OR, for example SOAP | WORK when the supported path must carry the entry both over SOAP and into Work instances. LOCAL alone confines propagation to the current thread; ONEWAY suppresses return propagation to the caller.
`PrimitiveContextFactory` Class	Convenience factory for built-in context values: create(String), create(long), createASCII(String), create(Serializable), and createMutable(Serializable). The String/long overloads create the corresponding built-in contexts. Serializable forms can throw IOException and are opaque to underlying protocols.

For the complete API documentation about context propagation, see the `weblogic.workarea` Javadocs.

Programming Context Propagation: Main Steps

You can associate information to a request on a client, retrieve that information on the server, and then retrieve the value updated by the server instance using context propagation.

The following procedure describes the high-level steps to use context propagation with WebLogic Server. This example demonstrates how to associate information to a request on a client, how to retrieve that information on the server, and then how to retrieve the value updated by the server instance. It is assumed in the procedure that you have already set up your iterative development environment and have an existing client and application that you want to update to use context propagation by using the `weblogic.workarea` API.

1. Update your client application to create the `WorkContextMap` and `WorkContext` objects and then add user data to the context.

   See Programming Context Propagation in a Client.

2. If your client application is standalone (rather than running in a Jakarta EE component deployed to WebLogic Server), ensure that its CLASSPATH includes the Jakarta EE application client, also called the *thin client*.

   See *Developing Stand-alone Clients for Oracle WebLogic Server*.

3. Update your application (EJB, Web service, servlet, and so on) to also create a `WorkContextMap` and then get the context and user data that you added from the client application.

   See Programming Context Propagation in an Application.

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

Programming Context Propagation in a Client

You can program context propagation to get “associated” user information when a client invokes an application.

The following sample Java code shows a standalone Java client that invokes a Web service; the example also shows how to use the `weblogic.workarea.*` context propagation APIs to associate user information with the invoke. The code relevant to context propagation is shown in bold and explained after the example.

For the complete API documentation about context propagation, see the `weblogic.workarea` Javadocs.

> **Note**
>
> See *Developing JAX-WS Web Services for Oracle WebLogic Server* for information on creating Web services and client applications that invoke them.

```
package examples.workarea.client;
import java.rmi.RemoteException;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Stub;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import weblogic.workarea.WorkContextMap;
import weblogic.workarea.WorkContext;
import weblogic.workarea.PrimitiveContextFactory;
import weblogic.workarea.PropagationMode;
import weblogic.workarea.PropertyReadOnlyException;
/**
 * This is a simple standalone client application that invokes the
 * the <code>sayHello</code> operation of your WorkArea Web service.
 *
 */
public class Main {
  public final static String SESSION_ID= "session_id_key";
  public static void main(String[] args)
      throws ServiceException, RemoteException, NamingException,
PropertyReadOnlyException{
    YourWorkAreaService service = new YourWorkAreaService(args[0] + "?WSDL");
    YourWorkAreaPortType port = service.getWorkAreaPort();
    WorkContextMap map = (WorkContextMap)new InitialContext().lookup("java:comp/
WorkContextMap");
    WorkContext stringContext = PrimitiveContextFactory.create("A String
Context");
    // Put a string context
    map.put(SESSION_ID, stringContext, PropagationMode.SOAP);
    try {
      String result = null;
      result = port.sayHello("Hi there!");
      System.out.println( "Got result: " + result );
    } catch (RemoteException e) {
      throw e;
    }
  }
}
```


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

In the preceding example:

- The following code shows how to import the needed `weblogic.workarea.*` classes, interfaces, and exceptions:

```
import weblogic.workarea.WorkContextMap;
import weblogic.workarea.WorkContext;
import weblogic.workarea.PrimitiveContextFactory;
import weblogic.workarea.PropagationMode;
import weblogic.workarea.PropertyReadOnlyException;
```

- Substitute your implementation of the `WorkArea` service and port for your Web service for *`YourWorkAreaService`* and *`YourWorkAreaPortType`*.
- The following code shows how to create a `WorkContextMap` by doing a JNDI lookup of the context propagation-specific JNDI name `java:comp/WorkContextMap`:

```
WorkContextMap map = (WorkContextMap)
      new InitialContext().lookup("java:comp/WorkContextMap");
```

- The following code shows how to create a `WorkContext` by using the `PrimitiveContextFactory`. In this example, the `WorkContext` consists of the simple String value `A String Context`. This String value is the user data that is passed to the invoked Web service.

```
WorkContext stringContext =
     PrimitiveContextFactory.create("A String Context");
```

- The following code saves the `stringContext` under the `SESSION_ID` key in the `WorkContextMap`. Specifying the propagation mode of `SOAP` causes the propagation of the `stringContext` along any `SOAP` message sent to servers supporting context propagation.

```
map.put(SESSION_ID, stringContext, PropagationMode.SOAP);
```

Programming Context Propagation in an Application

You can program context propagation to get the user data and other associated information when the applications are invoked.

The following sample Java code shows a simple Java Web service (JWS) file that implements a Web service. The JWS file also includes context propagation code to get the user data that is associated with the invoke of the Web service. The code relevant to context propagation is shown in bold and explained after the example.

For the complete API documentation about context propagation, see the `weblogic.workarea` Javadocs.

> **Note**
>
> See *Developing JAX-WS Web Services for Oracle WebLogic Server* for information on creating Web services and client applications that invoke them.

```
package examples.workarea;
import javax.naming.InitialContext;
// Import the Context Propagation classes
import weblogic.workarea;
import weblogic.workarea.WorkContextMap;
```


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

```
import weblogic.workarea.WorkContext;
import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.WLHttpTransport;
@WebService(name="WorkAreaPortType",
            serviceName="WorkAreaService",
             targetNamespace="http://example.org")
@WLHttpTransport(contextPath="workarea",
                 serviceUri="WorkAreaService",
                  portName="WorkAreaPort")
/**
 * This JWS file forms the basis of simple WebLogic
 * Web service with a single operation: sayHello
 *
 */
public class WorkContextAwareWebService {
  public final static String SESSION_ID = "session_id_key";
  @WebMethod()
  public String sayHello(String message) {
   try {
WorkContextMap map = (WorkContextMap) new InitialContext().lookup("java:comp/
WorkContextMap");
WorkContext localwc = map.get(SESSION_ID);
WorkContext modifiedLocalWC = PrimitiveContextFactory.create(localwc.get() + "
could be replaced by a new value...");
    map.put(SESSION_ID, newLocalWC, PropagationMode.SOAP);
    System.out.println("local context: " + localwc);
    System.out.println("sayHello: " + message);
    return "The server received message: " + message + ", with SESSION_ID: " + localwc;
   } catch (Throwable t) {
    return "error";
   }
  }
}
```

In the preceding example:

- The following code shows how to import the needed context propagation APIs; in this case, only the WorkContextMap and WorkContext interfaces are needed:

  ```
  import weblogic.workarea.WorkContextMap;
  import weblogic.workarea.WorkContext;
  ```

- The following code shows how to create a `WorkContextMap` by doing a JNDI lookup of the context propagation-specific JNDI name `java:comp/WorkContextMap`:

  ```
  WorkContextMap map = (WorkContextMap)
      new InitialContext().lookup("java:comp/WorkContextMap");
  ```

- The propagation mode is `SOAP` only, meaning that propagation occurs both to the server with the request and to the client with the response. The following code shows how the server instance could modify the `stringContext`:

  ```
  WorkContext modifiedLocalWC = PrimitiveContextFactory.create(localwc.get() + " could
  be replaced by a new value...");
  ```

- The following code replaces the work context with an updated value. When retrieving `SESSION_ID` on the client after the server returns the response, the value updated by the server is now present on the client.

  ```
  map.put(SESSION_ID, newLocalWC, PropagationMode.SOAP);
  ```


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

17

Programming Jakarta Mail with WebLogic Server

Learn how to program Jakarta Mail with WebLogic Server to add email capabilities to your WebLogic Server applications.
This chapter includes the following sections:

Overview of Using Jakarta Mail with WebLogic Server Applications

WebLogic Server includes the Jakarta Mail API version 1.5 reference implementation. Using the Jakarta Mail API, you can add email capabilities to your WebLogic Server applications. Jakarta Mail provides access from Java applications to Internet Message Access Protocol (IMAP)- and Simple Mail Transfer Protocol (SMTP)-capable mail servers on your network or the Internet. It does not provide mail server functionality; you must have access to a mail server to use Jakarta Mail.

Documentation for using the Jakarta Mail API is available at `https://jakarta.ee/specifications/mail/1.6/`. This section describes how you can use Jakarta Mail in the WebLogic Server environment.

The `weblogic.jar` file contains the following Jakarta Mail API packages:

- `jakarta.mail`
- `jakarta.mail.event`
- `jakarta.mail.internet`
- `jakarta.mail.search`

The `weblogic.jar` also contains the Java Activation Framework (JAF) package, which Jakarta Mail requires.

The `jakarta.mail` package includes providers for Internet Message Access protocol (IMAP) and Simple Mail Transfer Protocol (SMTP) mail servers. There is a separate POP3 provider for Jakarta Mail, which is not included in `weblogic.jar`. You can download the POP3 provider at `https://maven.java.net/content/repositories/releases/com/sun/mail/pop3` and add it to the WebLogic Server classpath if you want to use it.

Understanding Jakarta Mail Configuration Files

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

Jakarta Mail discovers transport/store providers from provider metadata and service-provider registration in the runtime. The supplied modules already register the protocols used in this chapter; ordinary use requires no provider-file edits. For an additional transport or store, supply

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

a Jakarta Mail 2.1-compatible provider implementation and its documented provider registration, and make those resources visible to the application using the documented WebLogic classloading procedure. MIME content-handler extensions use Jakarta Activation registrations. Retain the server-provided Mail/Activation API types when adding an implementation.

See §§5.1 and 7.3 of Jakarta Mail 2.1 provider registry and content-type extension model

Configuring Jakarta Mail for WebLogic Server

To configure Jakarta Mail for use in WebLogic Server, you create a mail session in the WebLogic Remote Console. This allows server-side modules and applications to access Jakarta Mail services with JNDI, using session properties you preconfigure for them.

For example, by creating a mail session, you can designate the mail hosts, transport and store protocols, and the default mail user in the WebLogic Remote Console so that modules that use Jakarta Mail do not have to set these properties. A configured mail resource centralizes the properties and authentication used by applications through JNDI; it is a jakarta.mail.Session configuration object, not an open mail connection or a promise that every lookup returns the same Java object.

To use the WebLogic Remote Console to create a mail session, see Configure Access to JavaMail in the *Oracle WebLogic Remote Console Online Help*.

Configure the mail host's required TLS and authentication settings before using the examples. For the SMTP STARTTLS example, set the following mail-session properties and replace the host placeholder with the actual mail-server name. STARTTLS must succeed before the example proceeds. If the server requires SMTP authentication, also set mail.smtp.auth=true and configure the mail-session username/password through the supported WebLogic mail-session credential fields.

```
mail.transport.protocol=smtp
mail.smtp.host=your-configured-smtp-host
mail.smtp.starttls.enable=true
mail.smtp.starttls.required=true
mail.smtp.ssl.checkserveridentity=true
```

The reader selects pop3s explicitly; replace the POP3 host placeholder and use the mail service's configured TLS listener/port.

```
mail.store.protocol=pop3s
mail.pop3s.host=your-configured-pop3-host
mail.pop3s.ssl.checkserveridentity=true
```

Use the actual host name matching the server certificate and a trust configuration that validates its issuer. Retain certificate and hostname verification. The examples do not put credentials in source or URLs.

Use the looked-up Session directly when its configured properties and authentication are appropriate. Session.getInstance is a static factory for an independent Session; invoking it through a session variable does not inherit the original Session or its Authenticator. If an application requires independent configuration, copy the original properties into a new Properties object, apply the overrides, and create the new Session using Session.getInstance(properties, authenticator), supplying the required Authenticator explicitly.

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

Copying properties alone does not copy the configured authenticator. Do not mutate the properties of a shared JNDI Session to customize one operation.

```
package examples.mail;

import java.util.Properties;
import jakarta.mail.Authenticator;
import jakarta.mail.Session;

public final class MailSessions {
    public static Session customize(Session configured, Properties overrides,
            Authenticator authenticator) {
        Properties properties = new Properties();
        properties.putAll(configured.getProperties());
        properties.putAll(overrides);
        return Session.getInstance(properties, authenticator);
    }
}
```

Sending Messages with Jakarta Mail

You can send a message using Jakarta Mail within a WebLogic Server module.

Here are the steps to send a message with Jakarta Mail:

Create and target a mail session whose JNDI name is myMailSession, or replace that literal with your configured name. Configure mail.from as an application-controlled sender authorized by the mail service, plus the SMTP host, TLS and any required authentication described in Configuring Jakarta Mail for WebLogic Server. Call send(to, subject, messageText) from your server-side code. Apply the application's recipient authorization rules before this call; syntax validation is not authorization.Configuring Jakarta Mail for WebLogic Server

```
package examples.mail;

import java.nio.charset.StandardCharsets;
import java.util.Date;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import jakarta.mail.Message;
import jakarta.mail.MessagingException;
import jakarta.mail.Multipart;
import jakarta.mail.Session;
import jakarta.mail.Transport;
import jakarta.mail.internet.InternetAddress;
import jakarta.mail.internet.MimeBodyPart;
import jakarta.mail.internet.MimeMessage;
import jakarta.mail.internet.MimeMultipart;

public final class SendMail {
    public static void send(String to, String subject, String messageText)
            throws NamingException, MessagingException {
        InitialContext naming = new InitialContext();
        try {
            Session session = (Session) naming.lookup("myMailSession");
            Transport.send(createMessage(session, to, subject, messageText));
```


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

```
            } finally {
                naming.close();
            }
        }

        public static MimeMessage createMessage(Session session, String to,
                String subject, String messageText) throws MessagingException {
            String from = session.getProperty("mail.from");
            if (from == null || from.isBlank()) {
                throw new MessagingException("Configure an application-controlled
mail.from address");
            }
            requireSingleLine(from);
            requireSingleLine(to);
            requireSingleLine(subject);
            InternetAddress sender = new InternetAddress(from, true);
            sender.validate();
            InternetAddress[] recipients = InternetAddress.parse(to, true);
            if (recipients.length == 0) {
                throw new MessagingException("Specify at least one recipient");
            }
            for (InternetAddress recipient : recipients) {
                recipient.validate();
            }
            MimeMessage message = new MimeMessage(session);
            message.setFrom(sender);
            message.setRecipients(Message.RecipientType.TO, recipients);
            message.setSubject(subject, StandardCharsets.UTF_8.name());
            message.setSentDate(new Date());
            MimeBodyPart text = new MimeBodyPart();
            text.setText(messageText, StandardCharsets.UTF_8.name());
            Multipart content = new MimeMultipart();
            content.addBodyPart(text);
            message.setContent(content);
            return message;
        }

        private static void requireSingleLine(String value) throws
MessagingException {
            if (value == null || value.indexOf('\r') >= 0 || value.indexOf('\n')
>= 0) {
                throw new MessagingException("Address and subject values must be
single-line text");
            }
        }
}
```

Transport.send submits the message to the configured transport; this does not guarantee delivery to the recipient's inbox. The caller handles NamingException and MessagingException.

Validate user-provided contact addresses and use them as Reply-To only when required by the application; keep the configured, authorized sender. The example closes its naming context and retains the one-part multipart construction.

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

Reading Messages with Jakarta Mail

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

The Jakarta Mail API allows you to connect to a message store, which could be an IMAP server or POP3 server. Messages are stored in folders. With IMAP, message folders are stored on the mail server, including folders that contain incoming messages and folders that contain archived messages. With POP3, the server provides a folder that stores messages as they arrive. The POP3 provider exposes the remote mailbox as INBOX and retrieves message data as required. Reading does not automatically persist a local mailbox or remove messages from the server..

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

The API provides several options for reading messages. See the Jakarta Mail API for more information.

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

getStore("pop3s") explicitly selects POP3 TLS. Open the folder before retrieving messages and copy needed data while it is open because content may be fetched lazily. The example returns copied subject strings. Store is AutoCloseable; the folder uses explicit close(false), which does not request expunge.

This example uses the configured myMailSession resource and reads subject headers from INBOX over POP3 TLS. Replace the JNDI literal if needed. Supply mailHost, username and password through the application’s approved configuration/credential handling; no sample password is embedded. Configure the pop3s settings described below before invoking it.

```
package examples.mail;

import java.util.ArrayList;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import jakarta.mail.Folder;
import jakarta.mail.Message;
import jakarta.mail.MessagingException;
import jakarta.mail.Session;
import jakarta.mail.Store;

public final class ReadMail {
    public static List<String> readSubjects(String mailHost, String username,
            String password) throws NamingException, MessagingException {
        InitialContext naming = new InitialContext();
        try {
            Session session = (Session) naming.lookup("myMailSession");
            try (Store store = session.getStore("pop3s")) {
                store.connect(mailHost, username, password);
                Folder inbox = store.getFolder("INBOX");
```


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

```
                inbox.open(Folder.READ_ONLY);
                try {
                    List<String> subjects = new ArrayList<>();
                    for (Message message : inbox.getMessages()) {
                        subjects.add(message.getSubject());
                    }
                    return subjects;
                } finally {
                    if (inbox.isOpen()) {
                        inbox.close(false);
                    }
                }
            }
        } finally {
            naming.close();
        }
    }
}
```

Reading messages from an IMAP server is similar to reading messages from a POP3 server. With IMAP, however, the Jakarta Mail API provides methods to create and manipulate folders and transfer messages between them. If you use an IMAP server, you can implement a full-featured, Web-based mail client with much less code than if you use a POP3 server. If the application needs local persistence or its own folder organization for POP3 messages, implement that storage explicitly or use a suitable local-store provider; the supplied POP3 provider does not create a persistent local message store. This chapter's read-only example does not mark messages for deletion and closes without expunging them.

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

18

Threading and Clustering Topics

Learn how to use threads in WebLogic Server as well as how to program applications for use in WebLogic Server clusters.

Using Threads in WebLogic Server

WebLogic Server is a sophisticated, multi-threaded application server and it carefully manages resource allocation, concurrency, and thread synchronization for the modules it hosts. To obtain the greatest advantage from WebLogic Server's architecture, construct your application modules according to the standard Jakarta EE APIs.

In most cases, avoid application designs that require creating new threads in server-side modules:

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

When independent repository queries can run concurrently, submit them as tasks to a container-provided jakarta.enterprise.concurrent.ManagedExecutorService. Use ManagedScheduledExecutorService for scheduled work. WebLogic Server supplies the default executor at java:comp/DefaultManagedExecutorService. The container manages the execution resources and associates supported application context with each task. Do not create an application-owned Thread or Executors pool as a substitute inside EJB business code; follow the threading restrictions of the component model. A managed thread factory is available where explicit managed threads are appropriate, subject to those component rules.

Bound concurrent work and configure the managed resource’s limits for the workload. Use completion/timeout handling appropriate to the request; avoid waiting while holding locks or other resources needed by the submitted tasks. Treat shared application state as concurrent state, and follow the component’s rules for resource access. Do not assume a task inherits the submitting transaction.

Applications submit work to the managed resource but do not own its lifecycle: do not call shutdown, shutdownNow or close on an injected/lookup managed executor. Make tasks respond to cancellation and interruption and release their resources.

Review the design for deadlocks and thread starvation, and test the managed concurrent workload under representative and overload conditions.

Unmanaged background threads can outlive an application deployment and retain its classloader, resources or application state, whether they are daemon or non-daemon threads. Daemon status controls JVM shutdown participation; it does not provide application lifecycle management. Use managed concurrency resources and make tasks honor interruption/ cancellation so that work can finish and release resources when the application stops or is redeployed.

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

Be sure to test multithreaded code under increasingly heavy loads, adding clients even to the point of failure. Observe the application performance and WebLogic Server behavior and then add checks to prevent failures from occurring in production.

See Configuring Concurrent Managed Objects in Administering Server Environments for Oracle WebLogic Server

Package this CDI bean in an application with annotated bean discovery enabled. Obtain it through container injection rather than new RepositoryTasks(), so the managed executor resource is injected. Pass a thread-safe Callable<String> implementing an independent repository query. submit returns a Future; its caller must handle completion, failure, cancellation and any required timeout. The container owns the executor; this bean does not create a thread pool or shut down the executor.

```
package examples.concurrent;

import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import jakarta.annotation.Resource;
import jakarta.enterprise.concurrent.ManagedExecutorService;
import jakarta.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class RepositoryTasks {
    @Resource(lookup = "java:comp/DefaultManagedExecutorService")
    private ManagedExecutorService executor;

    public Future<String> submit(Callable<String> query) {
        return executor.submit(query);
    }
}
```

Thread Scheduling APIs

WebLogic Server schedules concurrent work through its Work Manager infrastructure and provides Jakarta Concurrency services and CommonJ APIs for application programming.

WebLogic Server provides several interfaces for submitting and scheduling concurrent work.

- **WebLogic Work Manager APIs:** The `weblogic.work` package contains WebLogic-specific scheduling interfaces, including `weblogic.work.WorkManager` and WorkManagerFactory. `WorkManager.schedule(Runnable)` submits work to a Work Manager.
- **Jakarta Concurrency services:** Use `jakarta.enterprise.concurrent.ManagedExecutorService` to `submit` `Runnable` or `Callable` tasks, and `ManagedScheduledExecutorService` for delayed or periodic execution. The container manages execution resources, the supported application context and lifecycle. Methods such as `submit` return `Future` handles for completion and result handling. Use these standard services for new portable Jakarta EE concurrency code, subject to the component model's threading rules. See Using Threads in WebLogic Server above and Configuring Concurrent Managed Objects in Administering Server Environments for Oracle WebLogic Server.
- **CommonJ APIs:** The `commonj.work.WorkManager` API submits `commonj.work.Work` instances and returns `WorkItem` handles. Its `waitForAll` and `waitForAny` methods let a caller wait for submitted work items. The `commonj.timers.TimerManager` API provides timer scheduling. Existing CommonJ applications can continue to use these supported

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

APIs. See Using the Timer and Work Manager API in Developing CommonJ Applications for Oracle WebLogic Server.

Work Manager request classes and constraints control scheduling of associated server work. Managed concurrency services and CommonJ work submissions integrate with the WebLogic scheduling infrastructure; their application APIs and context/lifecycle contracts remain distinct. Concurrent execution can improve throughput or reduce response latency for suitable independent work, but results depend on resource limits, contention and workload.

For scheduling configuration, see Using Work Managers to Optimize Scheduled Work in Administering Server Environments for Oracle WebLogic Server.

Programming Applications for WebLogic Server Clusters

There are certain requirements and restrictions when you deploy JSPs and servlets, and EJBs to a WebLogic Server cluster. Also you need to understand the implications of binding clustered objects in the JNDI tree when you develop EJBs or custom RMI objects in a cluster.

JSPs and servlets that will be deployed to a WebLogic Server cluster must observe certain requirements for preserving session data. See Requirements for HTTP Session State Replication in *Administering Clusters for Oracle WebLogic Server* for more information.

EJBs deployed in a WebLogic Server cluster have certain restrictions based on EJB type. See Understanding WebLogic Enterprise JavaBeans in *Developing Jakarta Enterprise Beans Using Deployment Descriptors* for information about the capabilities of different EJB types in a cluster. EJBs can be deployed to a cluster by setting clustering properties in the EJB deployment descriptor.

If you are developing either EJBs or custom RMI objects for deployment in a cluster, also refer to Using WebLogic JNDI in a Clustered Environment in *Developing JNDI Applications for Oracle WebLogic Server* to understand the implications of binding clustered objects in the JNDI tree.

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

19

Developing OSGi Bundles for WebLogic Server Applications

Learn about the OSGi environment in WebLogic Server and how to deploy OSGi bundles to WebLogic Server. Developers who want to use OSGi in their applications can easily share OSGi facilities, such as the OSGi service registry, class loaders, and other OSGi services. For general information about OSGi, see https://www.osgi.org/.

Understanding OSGi

OSGi defines a modular, service-oriented component model for Java. Bundles declare package dependencies and versions, and communicate through an OSGi service registry. The OSGi Working Group at the Eclipse Foundation continues the mission of the former OSGi Alliance. See OSGi Working Group and the OSGi specification and Javadoc index.

See OSGi Working Group; OSGi specification and Javadoc index.

-
-

OSGi offers you the following benefits:

- Versioning of package wiring, for both implementors and users of interfaces.
- The "uses" directive allows for intelligent wiring of class loaders and helps ensure a consistent class space.
- Service and package access controls where the required security mechanisms are configured and supported by the Java runtime.
- Dynamic service wiring through an active registry.
- Various standard OSGi specifications provided by multiple vendors.

Features Provided in WebLogic Server OSGi Implementation

WebLogic Server allows you to configure and manage one or more instances of an OSGi framework. You can also create and deploy your own OSGi bundles.

WebLogic Server manages configured OsgiFrameworkMBean framework instances. An application can select one bundle for classloader and application-JNDI integration, as described in Accessing Deployed Bundle Objects From JNDI.Accessing Deployed Bundle Objects From JNDI

Applications can also deploy their own OSGi bundles. One specific OSGi bundle from the chosen framework instance can be used in the application classloader hierarchy.

WebLogic Server allows you to:

- Configure and manage one or more instances of an OSGi framework using WLST.

  WebLogic Server includes the Apache Felix implementation of the OSGi framework. See http://felix.apache.org for information on Felix.

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

- Create and deploy your own OSGi bundles.

  WebLogic Server includes an OSGi bundle containing the OSGi API. You can use this API to create your own OSGi bundles.
- One specific OSGi bundle from the chosen framework instance can be used in the application classloader hierarchy.
- Access OSGi bundles directly from JNDI.
- Deploy and undeploy OSGi bundles.
- Log OSGi status via the WebLogic Server logging mechanism.
- Incorporate the OSGi services of your choice.
- Manage OSGi bundle start levels for deployed bundles.

These topics are described in the sections that follow.

Configuring the OSGi Framework

WebLogic Server manages Apache Felix framework instances and their initialization properties.

An OSGi framework manages bundle installation, lifecycle and package wiring. WebLogic Server embeds and manages Apache Felix framework instances. Configure service availability and any required access controls for the deployed environment.

> **Note**
>
> WebLogic Server supports only the Felix framework. Other OSGi Frameworks are not supported and have not been tested.

Configuring OSGi Framework Instances

WebLogic Server includes an OSGi framework by default, but it does not automatically start it.

Configure and target an OSGi framework before use.

- Use WLST or a Java JMX client to create an OsgiFrameworkMBean in the domain edit configuration, assign its targets, save and activate the change. The resulting domain configuration is persisted in DOMAIN_HOME/config/config.xml.

In either case, configuration of an OSGi framework instance is controlled by the OsgiFrameworkMBean. For each framework associated with an OsgiFrameworkMBean, WebLogic Server boots an OSGi framework with a unique name.

You configure the OSGi framework attributes shown in Table 19-1.

Table 19-1 OSGi Framework Attributes

Attribute	Usage
Targets	TargetMBean[]; select one or more configured servers or clusters on which this framework is to run.
Name	String; the name of the OsgiFrameworkMBean in its containing domain. Use a unique framework name in that domain.

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

Table 19-1 (Cont.) OSGi Framework Attributes

Attribute	Usage
FactoryImplementationClass	String; FrameworkFactory implementation class; default org.apache.felix.framework.FrameworkFactory.
DeployInstallationBundles	String; legal values populate (default) and ignore. Controls loading bundles from WL_HOME/server/osgi-lib; it does not disable bundles packaged in applications.
DynamicallyCreated	Read-only boolean reporting whether the bean was dynamically created; this is not a persistence setting for the writer or administrator to set.
InitProperties	java.util.Properties passed to framework initialization, subject to WebLogic overrides described in Configuring OSGi Framework Persistence. Standard framework properties and implementation-specific properties may be supplied.
OrgOsgiFrameworkBootdelegation	The name of the `org.osgi.framework.bootdelegation` property. Note that this value, if set, will take precedence over anything specified in the init-properties.
OrgOsgiFrameworkSystemPackagesExtra	The name of the `org.osgi.framework.system.packages.extra` property. Note that this value, if set, will take precedence over anything specified in the init-properties.
RegisterGlobalDataSources	Boolean. Returns true if global data sources should be added to the OSGi service registry. Default: true.
RegisterGlobalWorkManagers	Boolean. Returns true if global work managers should be added to the OSGi service registry. Default: true.

See Apache Felix Framework Configuration Properties: Framework configuration properties

OSGi Framework Configuration in config.xml

Example 18-1 illustrates the persisted domain configuration for a framework named test-osgi-frame targeted to AdminServer. Create this configuration through the WLST procedure below. Do not paste the fragment into a running domain's config.xml. Multiple frameworks require distinct names in the containing domain. Activation applies supported dynamic changes; follow the activation result for any required restart.

Example 19-1 Persisted OSGi Framework Configuration

```
<osgi-framework>
       <name>test-osgi-frame</name>
       <target>AdminServer</target>
   </osgi-framework>
```

Configuring OSGi Framework Instance From WLST

Mode: WLST online configuration edit, connected to the Administration Server for this domain, in a fresh dedicated named edit session. Run this script with WLST after setting WLS_ADMIN_URL, WLS_USER_CONFIG, WLS_USER_KEY, WLS_TARGET_SERVER, WLS_OSGI_FRAMEWORK and WLS_OSGI_EDIT_SESSION to your values. WLS_ADMIN_URL is the configured t3s:// administration endpoint; configure its trusted certificate and hostname validation. The credential configuration/key pair must already exist, be protected, and authorize this edit. Use a new, unused edit-session name with no unrelated

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

pending changes and a new framework name. The script looks up an existing server target; it does not create servers or replace an existing framework.

```
WL_HOME/common/bin/wlst.sh create_framework.py
```

Save as create_framework.py. Use wlst.cmd on Windows.

Example 19-2 Configuring OSGi Framework Instance From WLST

```
# Online configuration edit; run with WLST, not CPython.
import os

admin_url = os.environ['WLS_ADMIN_URL']
user_config = os.environ['WLS_USER_CONFIG']
user_key = os.environ['WLS_USER_KEY']
server_name = os.environ['WLS_TARGET_SERVER']
framework_name = os.environ['WLS_OSGI_FRAMEWORK']
session_name = os.environ['WLS_OSGI_EDIT_SESSION']
if not admin_url.startswith('t3s://'):
    raise ValueError('WLS_ADMIN_URL must use the configured TLS administration endpoint')

connect(userConfigFile=user_config, userKeyFile=user_key, url=admin_url)
locked = False
activation_started = False
try:
    edit(session_name)
    startEdit(60000, 120000, 'true')
    locked = True
    cd('/')
    domain = cmo
    target = domain.lookupServer(server_name)
    if target is None:
        raise ValueError('No configured server named ' + server_name)
    if domain.lookupOsgiFramework(framework_name) is not None:
        raise ValueError('Framework already exists: ' + framework_name)
    framework = domain.createOsgiFramework(framework_name)
    framework.addTarget(target)
    framework.setDeployInstallationBundles('populate')
    framework.setOrgOsgiFrameworkSystemPackagesExtra(
        'javax.naming,javax.sql,weblogic.work')
    print('Framework: ' + framework.getName())
    print('Targets: ' + str([t.getName() for t in framework.getTargets()]))
    save()
    activation_started = True
    activate(block='true')
    locked = False
except:
    if locked and not activation_started:
        try:
            undo('true', 'y')
        finally:
            cancelEdit('y')
    elif activation_started:
        print('Inspect the activation task and named edit session before retrying; '
              'do not cancel another session or assume rollback completed.')
    raise
finally:
    disconnect()
```

save persists pending changes; activate applies them. If activation fails or the connection is lost during activation, inspect the activation task and this dedicated edit session before cleanup

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

or retry. Follow any reported restart requirement; do not assume that saved configuration alone proves a running framework.

Configuring OSGi Framework Instance from a Java Program

Mode: online configuration edit through the Administration Server's edit MBean server. This standalone client creates one new OSGi framework and targets it to one existing server by name. It uses the framework's default initialization settings. Run it in a terminal against a domain with no unrelated pending changes and with edit activity coordinated for this operation. The client requests an exclusive edit lock, refuses existing pending changes and refuses an existing framework name. It prompts for an authorized administrator user/password; credentials are not embedded in the source or passed on the command line. The endpoint must provide T3S with a certificate trusted by the client, and hostname verification must succeed. Substitute your actual host, TLS port, configured server name and new framework name in the command. An administration port, when configured, must be used as required by your domain.

Use the release-matched supported JDK and a full WebLogic installation with WL_HOME set to its wlserver directory. Compile and run from the directory containing OsgiFrameworkClient.java. On Windows, use a semicolon classpath separator.

```
javac -cp "$WL_HOME/server/lib/weblogic.jar" OsgiFrameworkClient.java
java -cp ".:$WL_HOME/server/lib/weblogic.jar" OsgiFrameworkClient
admin.example.com 9002 managed1 osgi-example
```

admin.example.com, 9002, managed1 and osgi-example are illustrative argument values, not a preconfigured domain. Configure client TLS trust for your domain before running; do not disable certificate/hostname verification to make the example connect.

Example 19-3 Creating and Targeting an OSGi Framework Using JMX

```
import java.io.Console;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;
import weblogic.management.configuration.DomainMBean;
import weblogic.management.configuration.OsgiFrameworkMBean;
import weblogic.management.configuration.ServerMBean;
import weblogic.management.jmx.MBeanServerInvocationHandler;
import weblogic.management.mbeanservers.edit.ActivationTaskMBean;
import weblogic.management.mbeanservers.edit.ConfigurationManagerMBean;
import weblogic.management.mbeanservers.edit.EditServiceMBean;

public final class OsgiFrameworkClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 4) {
            throw new IllegalArgumentException(
                    "Usage: OsgiFrameworkClient adminHost tlsPort targetServer frameworkName");
        }
        Console console = System.console();
        if (console == null) {
            throw new IllegalStateException("Run from a terminal to enter credentials");
        }
```


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

```
        String user = console.readLine("Administrator user: ");
        char[] password = console.readPassword("Password: ");
        if (user == null || password == null) {
            throw new IllegalStateException("Credential entry was cancelled");
        }
        JMXServiceURL url = new JMXServiceURL("t3s", args[0],
                Integer.parseInt(args[1]), "/jndi/" + EditServiceMBean.MBEANSERVER_JNDI_NAME);
        Map<String, Object> environment = new HashMap<>();
        environment.put(Context.SECURITY_PRINCIPAL, user);
        environment.put(Context.SECURITY_CREDENTIALS, new String(password));
        environment.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
                "weblogic.management.remote");
        Arrays.fill(password, '\0');
        try (JMXConnector connector = JMXConnectorFactory.connect(url, environment)) {
            MBeanServerConnection connection = connector.getMBeanServerConnection();
            EditServiceMBean edit = (EditServiceMBean)
                    MBeanServerInvocationHandler.newProxyInstance(connection,
                            new ObjectName(EditServiceMBean.OBJECT_NAME),
                            EditServiceMBean.class, false);
            configure(edit.getConfigurationManager(), args[2], args[3]);
        }
    }

    static void configure(ConfigurationManagerMBean manager,
            String serverName, String frameworkName) throws Exception {
        DomainMBean domain = manager.startEdit(30000, 120000, true);
        boolean cleanStart = false;
        try {
            if (manager.haveUnactivatedChanges()) {
                throw new IllegalStateException("Resolve existing pending edits before running");
            }
            cleanStart = true;
            ServerMBean server = domain.lookupServer(serverName);
            if (server == null) {
                throw new IllegalArgumentException("Unknown server: " + serverName);
            }
            if (domain.lookupOsgiFramework(frameworkName) != null) {
                throw new IllegalArgumentException("Framework already exists: " + frameworkName);
            }
            OsgiFrameworkMBean framework = domain.createOsgiFramework(frameworkName);
            framework.addTarget(server);
            System.out.println("Creating " + framework.getName() + " on " + server.getName());
            manager.save();
        } catch (Exception failure) {
            try {
                try {
                    if (cleanStart) manager.undoUnactivatedChanges();
                } finally {
                    manager.stopEdit();
                }
            } catch (Exception cleanup) {
                failure.addSuppressed(cleanup);
            }
            throw failure;
        }
        // After activation starts, do not cancel a possibly different editor's session.
        try {
            ActivationTaskMBean task = manager.activate(120000);
            System.out.println(task.getDetails());
            if (task.getState() != ActivationTaskMBean.STATE_COMMITTED) {
                throw new IllegalStateException("Activation did not commit", task.getError());
            }
```


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

```
        } catch (Exception failure) {
            System.err.println("Inspect the activation task and pending edits before retrying; "
                    + "saved changes may still be pending.");
            throw failure;
        }
    }
}
```

See Accessing WebLogic Server MBeans with JMX. Managing a Domain's Configuration with JMX.

WebLogic's MBeanServerInvocationHandler maps returned MBean ObjectNames to typed proxies. The client looks up the target by name, creates a framework, adds that target, saves, and waits for activation. It prints the proposed framework/target and the activation details. save alone does not activate a change. A non-committed activation or a lost connection during activation requires inspection of the activation task and pending edits before retrying; saved changes may remain pending. Follow any reported restart requirement. Before activation begins, errors undo this attempt's changes and release its edit lock; existing saved pending edits detected at the start are retained. The client closes its JMX connection on exit. It does not verify live bundle deployment or expose every OsgiFrameworkMBean option.

Parameter Required for Installing Bundles in the Framework

The OsgiFrameWorkMBean MBean `Deploy Installation Bundles` attribute controls whether or not bundles present in the `osgi-lib` directory (described later in this chapter in Deploying OSGi Bundles in the osgi-lib Directory) are actually installed into the framework.

The `Deploy Installation Bundles` parameter accepts the following values:

- `ignore` — None of the bundles in this directory are installed and started.
- `populate` — The bundles are installed and started if possible. This is the default. Furthermore, a few extra packages are added to the boot delegation classpath parameters in order to enable the bundles in the `osgi-lib` directory if they are not already there.

  It is not be considered a failure that causes the system to not boot if these bundles do not properly resolve and therefore cannot be started.

Configuring OSGi Framework Persistence

OSGi frameworks define persistent bundle state and autostart behavior. WebLogic Server recreates its managed framework cache on first initialization of each framework instance: it sets org.osgi.framework.storage to the WebLogic-selected cache directory and org.osgi.framework.storage.clean to onFirstInit. These values override corresponding InitProperties entries. Do not rely on setting standard Felix cache properties in InitProperties to retain installed bundles across WebLogic framework restarts. Keep required bundles in the configured installation directory or in application packaging so they can be installed again. WebLogic Server does not replicate or fail over this framework cache to another server.

OSGi specification and Javadoc index

Using OSGi Services

Import the packages required by the services your bundle uses, and install/enable the service-provider bundles where needed. An Import-Package declaration alone does not create a service. Obtain services from BundleContext, handle their dynamic availability, and release acquired services. Existing Package Admin and org.osgi.service.startlevel.StartLevel APIs are compatibility APIs; new introspection code can use Bundle.adapt with BundleWiring and

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

BundleStartLevel as appropriate. Service availability and permission checks depend on the framework configuration and Java runtime.

See the Apache Felix Tutorial Example 1, Service Event Listener Bundle for an example of creating a simple bundle that listens for OSGi service events.

Connecting to an OSGi Console

You can inspect a configured framework through the OSGi API, as shown in Accessing Deployed Bundle Objects From JNDI. An optional Felix shell provides interactive administration. This WebLogic Server 26.1 branch includes Apache Felix 7.0.5; use the release's third-party licensing information for the component inventory. The distribution retains the compatibility filename WL_HOME/server/lib/org.apache.felix.org.apache.felix.main.jar. Additional console bundles are not enabled merely by configuring the framework.

Use this optional procedure only in an isolated development environment. Felix Remote Shell provides no authentication or encrypted transport; a connected user can control the framework. Keep it bound to loopback and do not expose or forward its port. Use API introspection instead when an interactive shell is unnecessary.

1. Obtain the Felix Shell (org.apache.felix.shell) and Remote Shell (org.apache.felix.shell.remote) bundles from the Apache Felix HTTPS downloads page.
2. Before creating the development framework, set its InitProperties entries osgi.shell.telnet.ip=127.0.0.1 and osgi.shell.telnet.port=6666. Set DeployInstallationBundles to populate. Use this port for only one framework instance; assign distinct loopback ports to any others that load a remote-shell bundle.
3. Put the two bundle JARs in WL_HOME/server/osgi-lib before creating/starting that framework; this directory is shared by framework instances configured to populate it.
4. Create and target the framework using the WLST procedure. From the same host, connect with telnet 127.0.0.1 6666. help lists commands; ps lists bundle identities and states. End the local session and remove the optional shell bundles from the development configuration when finished; existing running frameworks require their normal lifecycle procedure to remove installed bundles.

See Apache Felix Remote Shell reference

```
from java.util import Properties
props = Properties()
props.setProperty('osgi.shell.telnet.ip', '127.0.0.1')
props.setProperty('osgi.shell.telnet.port', '6666')
framework.setInitProperties(props)
```

Apply distinct per-framework loopback ports to every framework that loads the shared shell bundles. Before save in the fresh-framework WLST example, insert the following while preserving any other required initialization entries:

- `help` lists all the available commands
- `ps` lists all bundles and what state they are in

Apache FelixApache Felix downloads

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

Creating OSGi Bundles

You use the OSGi API bundle that is located in `wlserver/server/lib/org.apache.felix.org.apache.felix.main.jar` to create your own OSGi bundle.

See the Apache Felix Tutorial Example 1, Service Event Listener Bundle for an example of creating a simple bundle. As described in this example, the `Import-Package` attribute of the manifest file informs the framework of the bundle's dependencies on external packages. All bundles with an activator must import `org.osgi.framework` because it contains the core OSGi class definitions.

Deploying OSGi Bundles

Deploy OSGi bundle JARs either through a configured framework's installation directory or by packaging them in an EAR or WAR with an osgi-framework-reference.

Preparing to Deploy an OSGi Bundle on a Target System

You can deploy OSGi bundles from inside a JAR, EAR, or WAR file, as appropriate for your application.

Before you do this, you must first specify which OSGi framework you want your bundle to use, and identity the bundle to WebLogic Server.

> **Note**
>
> If the OSGi framework instance you specify does not exist on the target server, the OSGi bundle fails to deploy.

How you do this depends on whether your bundle is inside a WAR file or an EAR file:

- WAR — The framework instance and bundle name must be in an element in the Web application's `weblogic.xml` deployment descriptor file.
- EAR — The framework instance and bundle name must be in an element in the application's `weblogic-application.xml` deployment descriptor file.

  If the EAR file contains WAR files, then the bundles inside the WAR files are deployed using the `weblogic.xml` deployment descriptor file from the embedded WAR files.

The sections that follow describe the required steps in detail.

For more information about WebLogic Server deployment descriptors, see *Deploying Applications to Oracle WebLogic Server*.

Preparing to Deploy Bundles as Enterprise Applications

Before you deploy your OSGi bundle, you must first:

1. Use either the *`DOMAIN_HOME`*`\config\config.xml` deployment descriptor file or WLST to add an entry for the OSGi framework, as described in Configuring OSGi Framework Instances.

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

2. In the EAR file that contains the OSGi bundle, add both the name of the OSGi framework and the name of the bundle itself to the `weblogic-application.xml` deployment descriptor file.

Example 19-1 shows an example of updating `config.xml` to add the OSGi framework used by the WebLogic Server.

Example 19-4 shows an example of updating `weblogic-application.xml` to add both the name of the OSGi framework and the name and location of the bundle.

Example 19-4 Adding the Framework and Bundle to weblogic-application.xml

```
<osgi-framework-reference>
    <name>test-osgi-frame</name>
    <application-bundle-symbolic-name>com.oracle.weblogic.test.client
</application-bundle-symbolic-name>
  <bundles-directory>rashi/osgi-lib</bundles-directory>
</osgi-framework-reference>
```

The stanza in Example 19-4 tells the WebLogic Server to attach to the OSGi framework named "test-osgi-frame" and to find the bundle in that server with the symbolic name `com.oracle.weblogic.test.client` in order to find classes from that OSGi framework.

Preparing to Deploy Bundles as Web Applications

Before you install your bundle as a WAR file, you must first:

1. Use either the *`DOMAIN_HOME`*`\config\config.xml` deployment descriptor file or WLST to add an entry for the OSGi framework, as described in Configuring OSGi Framework Instances.
2. Add both the name of the OSGi framework and the name of the bundle itself to the web application's `weblogic.xml` deployment descriptor file.

Example 19-1 shows an example of updating `config.xml` to add the OSGi framework used by the WebLogic Server.

Example 19-5 shows an example of updating `weblogic.xml` to add both the name of the OSGi framework and the name and location of the bundle.

Example 19-5 Adding the Framework and Bundle to weblogic.xml

```
<osgi-framework-reference>
    <name>test-osgi-frame</name>
    <application-bundle-symbolic-name>com.oracle.weblogic.test.client
</application-bundle-symbolic-name>
  <bundles-directory>rashi/osgi-lib</bundles-directory>
</osgi-framework-reference>
```

The stanza in Example 19-4 tells the WebLogic Server to attach to the OSGi framework named "test-osgi-frame" and to find the bundle in that server with the symbolic name `com.oracle.weblogic.test.client` in order to find classes from that OSGi framework.

Global Work Managers

Work Managers prioritize work based on rules you define and by monitoring actual run time performance statistics. This information is then used to optimize the performance of your application. See Using Work Managers to Optimize Scheduled Work in *Administering Server Environments for Oracle WebLogic Server*.

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

The OSGi implementation can take advantage of global work managers if the Register Global Work Managers MBean attribute is set to true, as described in Table 19-1.

You can determine which global work manager is in use from a Java application, as shown in Example 19-6.

Call names with the valid BundleContext supplied to your active bundle's activator, or obtained through the selected bundle described in Accessing Deployed Bundle Objects From JNDI. Configure and target a global Work Manager named GlobalScopedWorkManager and leave RegisterGlobalWorkManagers enabled; substitute your configured service name in the filter. An empty result means no matching visible service is available. The returned names are a snapshot, not a guarantee that a service remains registered. The bundle must import org.osgi.framework and weblogic.work; weblogic.work must be visible in the framework system packages, as supplied by the WLST example.

Example 19-6 Determining Global Work Managers

```
package examples.osgi;

import java.util.ArrayList;
import java.util.List;
import org.osgi.framework.BundleContext;
import org.osgi.framework.InvalidSyntaxException;
import org.osgi.framework.ServiceReference;
import weblogic.work.WorkManager;

public final class WorkManagerLookup {
    private WorkManagerLookup() { }

    public static List<String> names(BundleContext context)
            throws InvalidSyntaxException {
        List<String> names = new ArrayList<>();
        for (ServiceReference<WorkManager> reference :
                context.getServiceReferences(WorkManager.class,
                        "(Name=GlobalScopedWorkManager)")) {
            WorkManager manager = context.getService(reference);
            if (manager == null) {
                continue; // The service may have been unregistered.
            }
            try {
                names.add(manager.getName());
            } finally {
                context.ungetService(reference);
            }
        }
        return names;
    }
}
```

Global Data Sources

In WebLogic Server, you can configure database connectivity by configuring JDBC data sources and multi data sources and then targeting or deploying the JDBC resources to servers or clusters in your WebLogic domain, as described in WebLogic Server Data Sources in *Understanding Oracle WebLogic Server*.

The OSGi implementation can take advantage of global data sources if the Register Global Data Sources MBean attribute is set to true, as described in Table 19-1.

You can determine which global data source is in use from a Java application, as shown in Example 19-7.

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

This example assumes an Oracle Database data source named OsgiDS is configured, deployed and targeted to this server and RegisterGlobalDataSources is enabled. Substitute the configured data-source service name in the Name filter. The fixed SELECT 1 FROM DUAL query is Oracle-specific; use an appropriate fixed query for another database. Call readValues with a valid BundleContext. An empty list means no matching visible service was obtained; handle or propagate SQLException and InvalidSyntaxException at the calling component boundary. try-with-resources closes ResultSet, Statement and Connection on success or failure; the finally block separately releases the acquired OSGi service. Import org.osgi.framework and javax.sql in the bundle manifest and make javax.sql visible to the framework. java.sql and javax.sql are Java SE APIs. Do not close the DataSource itself.

Example 19-7 Determining Global Data Sources

```
package examples.osgi;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import javax.sql.DataSource;
import org.osgi.framework.BundleContext;
import org.osgi.framework.InvalidSyntaxException;
import org.osgi.framework.ServiceReference;

public final class DataSourceLookup {
    private DataSourceLookup() { }

    public static List<String> readValues(BundleContext context)
            throws InvalidSyntaxException, SQLException {
        List<String> values = new ArrayList<>();
        for (ServiceReference<DataSource> reference :
                context.getServiceReferences(DataSource.class, "(Name=OsgiDS)")) {
            DataSource dataSource = context.getService(reference);
            if (dataSource == null) {
                continue;
            }
            try {
                // Oracle Database example; use an appropriate query for other databases.
                try (Connection connection = dataSource.getConnection();
                     Statement statement = connection.createStatement();
                     ResultSet rows = statement.executeQuery("SELECT 1 FROM DUAL")) {
                    while (rows.next()) {
                        values.add(rows.getString(1));
                    }
                }
            } finally {
                context.ungetService(reference);
            }
        }
        return values;
    }
}
```


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

Deploying OSGi Bundles in the osgi-lib Directory

> **Note**
>
> OsgiFrameworkMBean.DeployInstallationBundles controls installation-directory bundles as described in Parameter Required for Installing Bundles in the Framework. The default value is populate, which attempts to install the bundles in WL_HOME/server/osgi-lib and start those that can be started.

To deploy a bundle with the start-level of 1, create the *WL_HOME*`/server/osgi-lib` directory if it does not already exist, and then copy the OSGi bundle JAR file to it.

The scanner attempts to install readable files with the case-sensitive .jar suffix. Each file must be a valid OSGi bundle JAR; files ending in .ear or .war are not scanned as bundles.

The directory is scanned when each configured framework is initialized, and it is not monitored afterward. To load a new directory bundle through this mechanism, reinitialize the affected framework through its managed lifecycle, for example by a controlled restart of its targeted WebLogic Server. Coordinate this with dependent applications.

Setting Bundle and Framework Start Levels

Bundle JARs directly in WL_HOME/server/osgi-lib are assigned start level 1. One-level subdirectories whose complete names parse as positive Java int values, for example 2 or 3, assign that bundle start level to their JARs. Names such as 3-startup and 0 are not valid numbered directories. A bundle whose start level is above the framework's active start level remains inactive until the framework reaches that level. The default beginning framework level is 1. To start bundles assigned level 3 during framework startup, set InitProperties entry org.osgi.framework.startlevel.beginning to the string 3 before that framework is initialized. Start levels control lifecycle ordering; they do not replace correct package/service dependencies.

For a new framework, optionally insert the following before save in the WLST script. For an existing framework preserve its other initialization entries.

```
from java.util import Properties
props = Properties()
props.setProperty('org.osgi.framework.startlevel.beginning', '3')
framework.setInitProperties(props)
```

Accessing Deployed Bundle Objects From JNDI

When an application's osgi-framework-reference selects an application-bundle-symbolic-name, WebLogic Server finds the matching bundle in that framework, attaches its class finder to the application, and binds that Bundle object at java:app/osgi/Bundle in the application's naming environment. An optional application-bundle-version constrains the version match; without a constraint, the highest matching bundle version is selected. This binding is not automatically created merely by starting a framework or deploying bundles without selecting an application bundle. The selected bundle must exist; a failed selection prevents application deployment. A caller can obtain its BundleContext while that selected bundle has an active context. Do not cache and reuse the context after the bundle stops.

Example 19-8 shows how to access a bundle that you create from JNDI.

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

Call describe() from a component of the deployed application whose reference selects a bundle. It returns a snapshot of installed bundles and registered-service metadata. Bundle states are the numeric Bundle constants; ACTIVE identifies an active bundle. The listing does not acquire service objects or change framework state. A stopped bundle can invalidate its context during inspection; handle NamingException and lifecycle failures at the component boundary and repeat inspection only while the selected bundle is active. Require framework visibility and bundle imports for org.osgi.framework, org.osgi.framework.startlevel and javax.naming. The WLST example exports javax.naming. Use WorkManagerLookup and DataSourceLookup to acquire service objects.

Example 19-8 Reading Bundle Metadata From Application JNDI

```
package examples.osgi;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceReference;
import org.osgi.framework.startlevel.BundleStartLevel;

public final class BundleInfo {
    private BundleInfo() { }

    public static List<String> describe() throws NamingException {
        InitialContext naming = new InitialContext();
        Bundle selected;
        try {
            selected = (Bundle) naming.lookup("java:app/osgi/Bundle");
        } finally {
            naming.close();
        }
        BundleContext context = selected.getBundleContext();
        if (context == null) {
            throw new IllegalStateException("The selected bundle has no active context");
        }
        return describe(context);
    }

    public static List<String> describe(BundleContext context) {
        List<String> lines = new ArrayList<>();
        for (Bundle bundle : context.getBundles()) {
            BundleStartLevel level = bundle.adapt(BundleStartLevel.class);
            lines.add("id=" + bundle.getBundleId()
                    + " name=" + bundle.getSymbolicName()
                    + " version=" + bundle.getVersion()
                    + " state=" + bundle.getState()
                    + " startLevel=" + (level == null ? "n/a" : level.getStartLevel()));
            ServiceReference<?>[] references = bundle.getRegisteredServices();
            if (references != null) {
                for (ServiceReference<?> reference : references) {
                    lines.add("  service=" + reference.getProperty(Constants.SERVICE_ID)
                            + " types=" + Arrays.toString((String[])
                                    reference.getProperty(Constants.OBJECTCLASS))
                            + " Name=" + reference.getProperty("Name"));
                }
            }
        }
```


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

```
        return lines;
    }
}
```

Using OSGi Logging Via WebLogic Server

With installation bundles enabled, the supplied OSGi bridge forwards log events to WebLogic Server logging.

The installation supplies org.apache.felix.log-1.3.0.jar and com.oracle.weblogic.osgi.logger.jar in WL_HOME/server/osgi-lib. With DeployInstallationBundles=populate, the framework attempts to install and start these bundles; the bridge listens to OSGi log events and forwards them to WebLogic logging. Setting ignore skips these installation-directory bundles, so this bridge is not automatically loaded by that mechanism.

The logger system name is `OSGiForApps`. The messages severity levels are mapped between OSGi and WebLogic Server as shown in Table 19-2.

The legacy integer constants are defined by org.osgi.service.log.LogService. The WebLogic severity constants are defined by weblogic.i18n.logging.Severities; this table describes the bridge's mapping of those integer log levels.

Table 19-2 OSGi and WebLogic Server Logging Severity Mapping

OSGi Severity Levels	WebLogic Server Severity Level
LogService.LOG_ERROR	Severities.ERROR
LogService.LOG_WARNING	Severities.WARNING
LogService.LOG_INFO	Severities.INFO
LogService.LOG_DEBUG	Severities.DEBUG

Configuring a Filtering ClassLoader for OSGi Bundles

A filtering classloader controls application package preference; configure bundle selection and package wiring separately.

Filtering controls whether matching application packages are preferred over parent packages. It does not install a dependency, export an OSGi package, or make an unavailable class visible. Select and attach the intended bundle through osgi-framework-reference before relying on its classes from the application. In an EAR, configure prefer-application-packages in META-INF/weblogic-application.xml at application scope. In a standalone WAR, configure it under container-descriptor in WEB-INF/weblogic.xml; keep prefer-web-inf-classes false when using this filter. For ordinary WAR libraries, package the JARs in WEB-INF/lib. For WAR-contained OSGi bundles, use WEB-INF/osgi-lib or the configured bundles directory. Bundle dependencies still require correct Import-Package/Export-Package wiring. OrgOsgiFrameworkSystemPackagesExtra can expose packages already available to the framework parent; it cannot supply an absent library. Export only the packages required by your integration. Keep Java SE javax.naming and javax.sql names unchanged; Jakarta EE APIs use their actual jakarta package names. See Using a Filtering ClassLoader in Chapter 12.Using a Filtering Classloader

```
<!-- Inside weblogic-application -->
<prefer-application-packages>
```


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

```
  <package-name>org.example.library.*</package-name>
</prefer-application-packages>


<!-- Inside weblogic-web-app for a standalone WAR -->
<container-descriptor>
  <prefer-web-inf-classes>false</prefer-web-inf-classes>
  <prefer-application-packages>
    <package-name>org.example.library.*</package-name>
  </prefer-application-packages>
</container-descriptor>
```

OSGI Example

WebLogic Server includes two simple example OSGi bundles: client and server. The server bundle (ServerBundle) exports a packet that the client bundle (ClientBundle) imports. The example produces an HTML page that displays the deployed OSGi bundles.

WebLogic Server includes an example that demonstrates how to deploy OSGi bundles to WebLogic Server. If you installed the WebLogic Server examples, the OSGi example source code is available in *ORACLE_HOME*`wl_server/examples/src/examples/osgi/osgiApp`, where *ORACLE_HOME* represents the directory in which the WebLogic Server code examples are configured. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

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

20

Using the WebSocket Protocol in WebLogic Server

WLS supports WebSocket protocol (RFC 6455), providing full-duplex communications between two peers over TCP. WLS implementation of WebSocket protocol enables development/ deployment of applications that communicate bidirectionally with clients. This implementation is mostly used to communicate with browsers running Web pages using W3C JavaScript WebSocket API. WLS implementation also supports Java clients.
This chapter includes the following sections:

Understanding the WebSocket Protocol

WebSocket is an application protocol that provides simultaneous two-way communication over a single TCP connection between a client and a server. The WebSocket protocol enables the client and the server to send data independently.

Browsers expose a JavaScript WebSocket API for connecting to endpoints and handling connection events. The browser API is defined by the WHATWG WebSockets Standard.WHATWG WebSockets Standard

For general wire-protocol information, see RFC 6455

Limitations of the HTTP Request-Response Model

In the traditional request-response model used in HTTP, the client requests resources and the server provides responses. The exchange is always initiated by the client; the server cannot send any data without the client requesting it first. This model worked well for the World Wide Web when clients made occasional requests for documents that changed infrequently, but the limitations of this approach are increasingly apparent as content changes quickly and users expect a more interactive experience on the web. The WebSocket protocol addresses these limitations by providing a full-duplex communication channel between the client and the server. Combined with other client technologies, such as JavaScript and HTML5, WebSocket enables web applications to deliver a richer user experience.

WebSocket Endpoints

In a WebSocket application, the server publishes a WebSocket **endpoint** and the client uses the endpoint's URI to connect to the server.

A WebSocket endpoint is represented by a URI in one of the following formats:

```
ws://host:port/path?query
wss://host:port/path?query
```

The `ws` scheme represents an unencrypted WebSocket connection.

The `wss` scheme represents an encrypted WebSocket connection.

The remaining components in these formats are as follows:

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

host
The host as defined in [RFC3986], Section 3.2.2.

port
Optional. The port as defined in [RFC3986], Section 3.2.3. The default port number is 80 for unencrypted connections and 443 for encrypted connections.

path
The path as defined in [RFC3986], Section 3.3. In a WebSocket endpoint, the path indicates the location of the endpoint within a server.

query
Optional. A query as defined in [RFC3986], Section 3.4.

Handshake Requests in the WebSocket Protocol

To initiate a WebSocket connection, the client sends a handshake request to a WebSocket endpoint that the server has published. The client locates the endpoint by using the end point's URI. The connection is established if the handshake request passes validation, and the server accepts the request. The handshake is compatible with existing HTTP-based infrastructure: web servers interpret the handshake as an HTTP connection upgrade request.

Example 20-1 Handshake Request from a WebSocket Client

The following example shows a handshake request from a client.

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

Example 20-2 Server Response to a Handshake Request from a WebSocket Client

The following example shows a handshake from a server in response to a handshake request from a client.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

The server applies a known operation to the value of the `Sec-WebSocket-Key` header to generate the value of the `Sec-WebSocket-Accept` header. The client applies the same operation to the value of the `Sec-WebSocket-Key` header. If the result matches the value received from the server, the connection is established successfully. The client and the server can send messages to each other after a successful handshake.

Messaging and Data Transfer in the WebSocket Protocol

The WebSocket protocol is symmetrical after the connection has been established: the client and the WebLogic Server instance can send messages to each other at any time while the connection is open, and they can close the connection at any time. Typically, clients connect to only one server, but servers accept connections from multiple clients.

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

WebSocket supports text messages (encoded as UTF-8) and binary messages. The control frames in WebSocket are *close*, *ping*, and *pong* (a response to a *ping* frame). Ping and pong frames may also contain application data.

Understanding the WebLogic Server WebSocket Implementation

WebLogic Server 26.1 supports Jakarta WebSocket 2.2 through its integrated Tyrus implementation.

For more information about the Jakarta WebSocket, see Jakarta WebSocket 2.2 specification.

Earlier versions of the API were developed as JSR 356.

> **Note**
>
> The proprietary weblogic.websocket and weblogic.websocket.annotation APIs, deprecated since 12.1.3, are retained for compatibility in this release. For current development and migration, use Jakarta WebSocket 2.2 with jakarta.websocket and jakarta.websocket.server types. Follow the current release documentation when assessing future support; this chapter does not guarantee compatibility with every future release.
>
> Although the Jakarta WebSocket coexists with the proprietary WebLogic Server WebSocket API, an application cannot contain calls to both APIs. Only one of the APIs can be used in an application.
>
> Information about how to use the deprecated API is available in the documentation for Oracle WebLogic Server 12c (12.1.2) in Chapter 17, Using WebSockets in WebLogic Server in *Developing Applications for Oracle WebLogic Server 12c (12.1.2)*.

The WebLogic Server WebSocket implementation includes the following components:

WebSocket Protocol Implementation

The WebSocket protocol implementation in WebLogic Server is provided by the reference implementation of the Jakarta WebSocket specification. This implementation of the WebSocket protocol handles connection upgrades, establishes and manages connections, and handles exchanges with the client.

WebLogic WebSocket Java API

The WebLogic WebSocket API is provided by the reference implementation of Jakarta WebSocket. This API consists of the following packages:

`jakarta.websocket.server`
This package contains annotations, classes, and interfaces to create and configure server endpoints.

`jakarta.websocket`
This package contains annotations, classes, interfaces, and exceptions that are common to client and server endpoints.

The API reference documentation for these packages is available in the following sections of the *Jakarta WebSocket 2.2 API documentation*:

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

- Package jakarta.websocket
- Package jakarta.websocket.server

Protocol Fallback for WebSocket Messaging

Protocol fallback provides a mechanism for using an alternative transport for WebSocket messaging when the WebSocket protocol is not supported. Typically the WebSocket protocol is not supported either because the WebSocket object is not available or because WebSocket frames are blocked by a firewall. In this release, the only supported alternative transport is HTTP Long Polling.

Protocol fallback enables you to rely on standard programming APIs to perform WebSocket messaging regardless of whether or not the runtime environment supports the WebSocket protocol. For more information, see Enabling Protocol Fallback for WebSocket Messaging.

Sample WebSocket Applications

If the WebLogic Server Examples component is installed and configured on your machine, you can use the WebSocket examples to demonstrate using WebSockets in WebLogic Server. For more information about running these examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Overview of Creating a WebSocket Application

Jakarta WebSocket enables you to create, configure, and deploy WebSocket endpoints in web applications. The WebSocket client API specified in JSR-356 also enables you to access remote WebSocket endpoints from any Java application.

The process for creating and deploying a WebSocket endpoint is as follows:

1. Create an endpoint class.
2. Implement the lifecycle methods of the endpoint.
3. Add your business logic to the endpoint.
4. Deploy the endpoint inside a web application.

Creating an Endpoint

By default the container creates a distinct endpoint instance for each peer connection. Endpoint state can therefore be kept in instance fields for that connection. Jakarta EE component lifecycles or a custom Configurator can change instance management; do not assume a shared instance is confined to one peer.

The Jakarta API for WebSocket enables you to create the following kinds of endpoints:

- Annotated endpoints
- Programmatic endpoints

The process is different for programmatic endpoints and annotated endpoints. In most cases, it is easier to create and deploy an annotated endpoint than a programmatic endpoint.

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

> **Note**
>
> By default the container creates a distinct endpoint instance for each peer connection. Endpoint state can therefore be kept in instance fields for that connection. Jakarta EE component lifecycles or a custom Configurator can change instance management; do not assume a shared instance is confined to one peer.

Creating an Annotated Endpoint

Creating an annotated endpoint enables you to handle life cycle events for a WebSocket connection by annotating methods of the endpoint class. For more information, see Handling Life Cycle Events in an Annotated WebSocket Endpoint. An annotated endpoint is deployed automatically with the application.
The Jakarta API for WebSocket enables you to create annotated server endpoints and annotated client endpoints.

To create an annotated server endpoint:

1. Write a Plain Old Java Object (POJO) class to represent the server endpoint.

   The class must have a public no-argument constructor.

2. Annotate the class declaration of the POJO class with the `jakarta.websocket.server.ServerEndpoint` annotation.

   This annotation denotes that the class represents a WebSocket server endpoint.

3. Set the value element of the `ServerEndpoint` annotation to the relative path to which the endpoint is to be deployed.

   The path must begin with a forward slash (/).

Example 20-3 Declaring an Annotated Server Endpoint Class

The following example shows how to declare an annotated server endpoint class. For an example of how to declare a programmatic endpoint class to represent the same endpoint, see Example 20-5.

This example declares the annotated server endpoint class `EchoEndpoint`. The endpoint is to be deployed to the /echo path relative to the application.

```
import jakarta.websocket.server.ServerEndpoint;
...
@ServerEndpoint("/echo")
public class EchoEndpoint {
...
}
```

Example 20-4 Declaring an Annotated Client Endpoint Class

To create an annotated client endpoint:

1. Write a Plain Old Java Object (POJO) class to represent the client endpoint.

   The class can have a constructor that takes arguments. However, to connect such an endpoint to a server endpoint, you must use the variant of the connectToServer method that takes an instance. You cannot use the variant that takes a class. For more information, see Connecting a Jakarta WebSocket Client to a Server Endpoint

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

2. Annotate the class declaration of the POJO class with the `jakarta.websocket.ClientEndpoint` annotation.

   This annotation denotes that the class represents a WebSocket client endpoint.

The following example shows how to declare an annotated client endpoint class.

This example declares the annotated client endpoint class `ExampleEndpoint`.

```
package examples.websocket;
import java.net.URI;
import jakarta.websocket.ContainerProvider;
import jakarta.websocket.Session;
import jakarta.websocket.WebSocketContainer;

public class ClientExample {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) throw new IllegalArgumentException("Supply the full ws://
or wss:// endpoint URI");
        URI uri = URI.create(args[0]);
        if (!"ws".equals(uri.getScheme()) && !"wss".equals(uri.getScheme())) {
            throw new IllegalArgumentException("Expected a ws or wss URI");
        }
        WebSocketContainer container = ContainerProvider.getWebSocketContainer();
        try (Session session = container.connectToServer(ExampleEndpoint.class, uri)) {
            System.out.println("Connected: " + session.getId());
            // Perform application-specific work here before this session is closed.
        }
    }
}
```

Creating a Programmatic Endpoint

Creating a programmatic endpoint requires you to handle life cycle events for a WebSocket connection by overriding methods of the endpoint's superclass. For more information, see Handling Life Cycle Events in a Programmatic WebSocket Endpoint. A programmatic endpoint is **not** deployed automatically with the application. You must deploy the endpoint explicitly. For more information, see Specifying the Path Within an Application to a Programmatic Endpoint.

To create a programmatic endpoint, extend the `jakarta.websocket.Endpoint` class.

Example 20-5 shows how to declare a programmatic endpoint class. For an example of how to declare an annotated endpoint class to represent the same endpoint, see Example 20-3.

Example 20-5 Declaring a Programmatic Endpoint Class

This example declares the programmatic endpoint class `EchoEndpoint`. For an example that shows how to specify the path within an application to this endpoint, see Example 20-6.

```
package examples.websocket;
import java.io.IOException;
import jakarta.websocket.Endpoint;
import jakarta.websocket.EndpointConfig;
import jakarta.websocket.MessageHandler;
import jakarta.websocket.Session;

public class ProgrammaticEcho extends Endpoint {
    @Override
    public void onOpen(Session session, EndpointConfig config) {
        session.addMessageHandler(String.class, (MessageHandler.Whole<String>) message -
> {
            try {
```


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

```
                session.getBasicRemote().sendText(message);
            } catch (IOException failure) {
                throw new java.io.UncheckedIOException(failure);
            }
        });
    }
}
```

Specifying the Path Within an Application to a Programmatic Endpoint

To enable remote clients to connect to a programmatic endpoint, you must specify the path within an application to the endpoint.

To specify the path within an application to a programmatic endpoint:

- Create the ServerEndpointConfig with the endpoint class and path, and return it from ServerApplicationConfig.getEndpointConfigs during deployment. Creating the configuration object alone does not deploy it.

When you deploy your application, the endpoint is available at the following URI:

`ws://host:port/context-root/path`

The replaceable items in this URI are as follows:

host
The host on which the application is running.

port
The port on which WebLogic Server listens for client requests.

context-root
the deployed web application's configured context root, which can differ from its deployment name

path
The path that you specified in the invocation of the `create` method.

For example, the URI to the endpoint at the `/echo` path relative to the `/echoapp` application running on the local host is `ws://localhost:8890/echoapp/echo`.

Example 20-6 shows how to register the programmatic endpoint during application startup.

Package both public classes in the WAR, under WEB-INF/classes or a JAR in WEB-INF/lib. ServerApplicationConfig is discovered during application startup and supplies the endpoint configuration. Returning the discovered annotated classes preserves their normal discovery. Do not package the annotated /echo example and this programmatic /echo example as competing mappings in the same WAR; they are alternatives.

Example 20-6 Registering a Programmatic Endpoint at Application Startup

This example specifies /echo as the path within an application to the programmatic endpoint ProgrammaticEcho from .

```
package examples.websocket;
import java.util.Set;
import jakarta.websocket.Endpoint;
import jakarta.websocket.server.ServerApplicationConfig;
import jakarta.websocket.server.ServerEndpointConfig;

public class EchoApplication implements ServerApplicationConfig {
```


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

```
    @Override
    public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends Endpoint>>
found) {
        return Set.of(ServerEndpointConfig.Builder.create(ProgrammaticEcho.class, "/
echo").build());
    }
    @Override
    public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> found) {
        return found;
    }
}
```

Handling Life Cycle Events for a WebSocket Connection

Different life cycle events for a WebSocket connection such as connection opened, message received, error, and connection closed are handled differently in an annotated endpoint and a programmatic endpoint.

Handling Life Cycle Events in an Annotated WebSocket Endpoint

Handling a life cycle event in an annotated WebSocket involves the following tasks:

1. Adding a method to your endpoint class to handle the event

   The allowed method parameters are defined by the annotation that you will use to designate the event.

2. Annotating the method declaration with the annotation that designates the event that the method is to handle.

lists the life cycle events in a WebSocket endpoint and the annotations available in the `jakarta.websocket` package to designate the methods that handle them. The examples in the table show the most common parameters for these methods. A jakarta.websocket.Session parameter is optional on lifecycle methods where permitted by the corresponding annotation. The table shows representative signatures; consult each annotation's allowed parameter combinations. A Session object represents a conversation between a pair of WebSocket endpoints.

For details about the combinations of parameters that are allowed by an annotation, see the API reference documentation for the annotation.

Table 20-1 Annotations in jakarta.websocket for WebSocket Endpoint Lifecycle Events

Event	Annotation	Example

Connection opened	`OnOpen`	`@OnOpen`
		`public void open(Session session,`
		`EndpointConfig conf)`
		`{ }`

Message received	`OnMessage`	@OnMessage public String message(String msg) { return msg; }

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

Table 20-1 (Cont.) Annotations in jakarta.websocket for WebSocket Endpoint Lifecycle Events

Event	Annotation	Example

Error	OnError	`@OnError`
		`public void error(Session session,`
		`Throwable error) { }`

Connection closed	OnClose	`@OnClose`
		`public void close(Session session,`
		`CloseReason reason)`
		`{ }`

Handling a Connection Opened Event

Handle a connection opened event to notify users that a new WebSocket conversation has begun.

To handle a connection opened event, annotate the method for handling the event with the `OnOpen` annotation.

Example 20-7 shows how to handle a connection opened event.

Example 20-7 Handling a Connection Opened Event

This example prints the identifier of the session when a WebSocket connection is opened.

```
import javax.websocket.OnOpen;
import javax.websocket.Session;
...
   @OnOpen
   public void openedConnection (Session session) {
      System.out.println("WebSocket opened: " + session.getId());
   }
...
```

Handling a Message Received Event

The Jakarta API for WebSocket enables you to handle the following types of incoming messages:

- Text messages
- Binary messages
- Pong messages

1. Add a method to your endpoint class to handle the type of the incoming message.

   Ensure that the data type of the parameter for receiving the message is compatible with the type of the message as shown in the following table.

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

Message Type	Data Type of the Parameter for Receiving the Message
Text	Any one of the following data types depending on how the message is to be received: • To receive the whole message: `java.lang.String` • To receive the whole message converted to a Java primitive or class equivalent to that type: the primitive or class equivalent • To receive the message in parts: `String` and boolean pair • To receive the whole message as a blocking stream: `java.io.Reader` • To receive the message encoded as a Java object: any type for which the endpoint has a text decoder (`jakarta.websocket.Decoder.Text` or `jakarta.websocket.Decoder.TextStream`)
Binary	Any one of the following data types depending on how the message is to be received: • To receive the whole message: byte array or `java.nio.ByteBuffer` • To receive the message in parts: byte array and boolean pair, or `ByteBuffer` and boolean pair • To receive the whole message as a blocking stream: `java.io.InputStream` • To receive the message encoded as a Java object: any object type for which the endpoint has a binary decoder (`jakarta.websocket.Decoder.Binary` or `jakarta.websocket.Decoder.BinaryStream`)
Pong	`jakarta.websocket.PongMessage`

2. Annotate the method declaration with the `OnMessage` annotation.

   You can have at most three methods annotated with `@OnMessage` in an endpoint, one method for each message type: text, binary, and pong.

> **Note**
>
> For an annotated endpoint, you add methods for handling incoming messages to your endpoint class. You are not required to create a separate message handler class. A programmatic endpoint registers a MessageHandler instance; it can be implemented by the endpoint class, a separate class, an anonymous class, or a lambda used with a typed addMessageHandler overload.
>
> To compare how to handle incoming messages for an annotated endpoint and a programmatic endpoint, see Example 20-8 and Example 20-12.

Example 20-8 Handling Incoming Text Messages for an Annotated Endpoint

The following example shows how to handle incoming text messages for an annotated endpoint.

This example replies to every incoming text message by sending the message back to the peer of this endpoint. The method that is annotated with the `OnMessage` annotation is a method of the endpoint class, not a separate message handler class.

For an example of how to perform the same operation for a programmatic endpoint, see Example 20-12.

```
import java.io.IOException;

import jakarta.websocket.OnMessage;
```


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

```
import jakarta.websocket.Session;
...
   @OnMessage
   public String onMessage(String msg) throws IOException {
      return msg;
   }
...
```

Example 20-9 Handling all Types of Incoming Messages

This example handles incoming text messages, binary messages, and pong messages. Text messages are received whole as `String` objects. Binary messages are received whole as `ByteBuffer` objects.

Place the following callback methods and hexadecimal helper inside the same annotated endpoint class, with the imports shown.

```
import java.nio.ByteBuffer;
import java.util.HexFormat;

import jakarta.websocket.OnMessage;
import jakarta.websocket.PongMessage;
import jakarta.websocket.Session;
...
   @OnMessage
   public void textMessage(Session session, String msg) {
      System.out.println("Text message: " + msg);
   }
   @OnMessage
   public void binaryMessage(Session session, ByteBuffer msg) {
      System.out.println("Binary message: " + hex(msg));
   }
   @OnMessage
   public void pongMessage(Session session, PongMessage msg) {
      System.out.println("Pong message: " +
                         hex(msg.getApplicationData()));
   }

   private static String hex(java.nio.ByteBuffer source) {
    java.nio.ByteBuffer copy = source.duplicate();
    byte[] bytes = new byte[copy.remaining()];
    copy.get(bytes);
    return java.util.HexFormat.of().formatHex(bytes);
}

...
```

The helper reads a duplicate so it does not change the original buffer position.

Handling an Error Event

You need to handle only error events that are not modeled in the WebSocket protocol, for example:

- Connection problems
- Runtime errors from message handlers
- Conversion errors in the decoding of messages

To handle an error event, annotate the method for handling the event with the `OnError` annotation.

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

Example 20-10 shows how to handle an error event.

Example 20-10 Handling an Error Event

This example prints a stack trace in response to an error event.

```
import jakarta.websocket.OnError;
import jakarta.websocket.Session;

   ...
   @OnError
   public void error(Session session, Throwable t) {
      t.printStackTrace();
      ...
   }
```

Handling a Connection Closed Event

You need handle a connection closed event only if you require some special processing before the connection is closed, for example, retrieving session attributes such as the ID, or any application data that the session holds before the data becomes unavailable after the connection is closed.

To handle a connection closed event, annotate the method for handling the event with the `OnClose` annotation.

Example 20-11 shows how to handle a connection closed event.

Example 20-11 Handling a Connection Closed Event

This example prints the message `Someone is disconnecting...` in response to a connection closed event.

```
import javax.websocket.OnClose;
import javax.websocket.Session;
...
  @OnClose
  public void bye(Session remote) {
    System.out.println("Someone is disconnecting...");
  }
...
```

Handling Life Cycle Events in a Programmatic WebSocket Endpoint

Table 20-2 summarizes how to handle lifecycle events in a programmatic WebSocket endpoint.

Table 20-2 Handling Life Cycle Events in a Programmatic WebSocket Endpoint

Event	How to Handle
Connection opened	Override the abstract `onOpen` method of the `Endpoint` class.
Message received	1. Implement MessageHandler.Whole<T> or MessageHandler.Partial<T>, or supply a matching typed lambda. 2. In onOpen, register it on that Session using addMessageHandler, preferably specifying the message Class for lambdas/ generic arrangements. 3. Put message processing in its onMessage implementation.
Error	Optional: Override the `onError` method of the `Endpoint` class. If you do not override this method, the `onError` method that your endpoint inherits from the `Endpoint` class is called when an error occurs.

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

Table 20-2 (Cont.) Handling Life Cycle Events in a Programmatic WebSocket Endpoint

Event	How to Handle
Connection closed	Optional: Override the `onClose` method of the `Endpoint` class. If you do not override this method, the `onClose` method that your endpoint inherits from the `Endpoint` class is called immediately before the connection is closed.

Example 20-12 shows how handle incoming text messages for a programmatic endpoint by handling connection opened events and message received events.

Example 20-12 Handling Incoming Text Messages for a Programmatic Endpoint

This example echoes every incoming text message. The example overrides the `onOpen` method of the `Endpoint` class, which is the only abstract method of this class.

The `Session` parameter represents a conversation between this endpoint and the remote endpoint. The `addMessageHandler` method registers message handlers, and the `getBasicRemote` method returns an object that represents the remote endpoint.

The message handler is implemented as an anonymous inner class. The `onMessage` method of the message handler is invoked when the endpoint receives a text message.

For more information about sending a message, see Sending a Message.

For an example of how to perform the same operation for an annotated endpoint, see Example 20-8.

```
package examples.websocket;
import java.io.IOException;
import jakarta.websocket.Endpoint;
import jakarta.websocket.EndpointConfig;
import jakarta.websocket.MessageHandler;
import jakarta.websocket.Session;

public class ProgrammaticEcho extends Endpoint {
    @Override
    public void onOpen(Session session, EndpointConfig config) {
        session.addMessageHandler(String.class, (MessageHandler.Whole<String>) message -
> {
            try {
                session.getBasicRemote().sendText(message);
            } catch (IOException failure) {
                throw new java.io.UncheckedIOException(failure);
            }
        });
    }
}
```

Register ProgrammaticEcho through EchoApplication at application startup as shown in Example 19-6.

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

Defining, Injecting, and Accessing a Resource for a WebSocket Endpoint

The Jakarta API for WebSocket allows you to use Contexts and Dependency Injection (CDI) to inject and access a resource that a WebSocket endpoint requires. You can use the injected resource from within a method for handling a lifecycle event for a WebSocket connection.

For more information about CDI, see Using Contexts and Dependency Injection for the Jakarta EE Platform.

To define, inject, and access a resource for a WebSocket endpoint:

1. Define a managed bean to represent the resource to inject.

   For more information, see Defining a Managed Bean.

2. In the endpoint class, inject the managed bean.

   For more information, see Injecting a Bean.

3. From within the relevant method, invoke methods of the injected bean as required.

The following examples show how to define, inject, and access a resource for a WebSocket endpoint:

- Example 20-13
- Example 20-14

Package these two classes in the same CDI-enabled WAR with annotated bean discovery. @Dependent makes InjectedSimpleBean a discoverable bean. WebLogic creates/manages the endpoint and performs injection; do not instantiate the endpoint yourself for this example. The bean's PostConstruct runs before the injected bean is used.

Example 20-13 Defining a Managed Bean for a WebSocket Endpoint

This example defines the managed bean class `InjectedSimpleBean`.

```
package examples.websocket;
import jakarta.annotation.PostConstruct;
import jakarta.enterprise.context.Dependent;

@Dependent
public class InjectedSimpleBean {
    private String text;
    @PostConstruct
    public void initialize() {
        text = " (from your server)";
    }
    public String getText() {
        return text;
    }
}
```

Example 20-14 Injecting and Accessing a Resource for a WebSocket Endpoint

This example injects an instance of the `InjectedSimpleBean` managed bean class into the server endpoint `SimpleEndpoint`. When the endpoint receives a message, it invokes the `getText` method on the injected bean. The method returns the text `(sent from your server)`. The endpoint then sends back a message which is a concatenation of the original message and gathered data.

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

The `InjectedSimpleBean` managed bean class is defined in Example 20-13.

```
package examples.websocket;
import jakarta.inject.Inject;
import jakarta.websocket.OnMessage;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint("/simple")
public class SimpleEndpoint {
    @Inject
    private InjectedSimpleBean bean;
    @OnMessage
    public String echo(String message) {
        return message + bean.getText();
    }
}
```

Sending a Message

The Jakarta API for WebSocket enables you to send text messages, binary messages, and ping frames from an endpoint to its connected peers.

- Text messages
- Binary messages
- Ping frames

Sending a Message to a Single Peer of an Endpoint

To send a message to a single peer of an endpoint:

1. Obtain the `Session` object from the connection.

   The `Session` object is available as a parameter in the lifecycle methods of the endpoint. How to obtain this object depends on whether the message that you are sending is a response to a message from a peer.

   - If the message is a response, obtain the `Session` object from inside the method that received the message.
   - If the message is **not** a response, store the `Session` object as an instance variable of the endpoint class in the method for handling a connection opened event. Storing the `Session` object in this way enables you to access it from other methods.

2. Use the `Session` object to obtain an object that implements one of the subinterfaces of `jakarta.websocket.RemoteEndpoint`.

   - If you are sending the message synchronously, obtain a `RemoteEndpoint.Basic` object. This object provides blocking methods for sending a message.

     To obtain a `RemoteEndpoint.Basic` object, invoke the `Session.getBasicRemote()` method.

   - If you are sending the message asynchronously, obtain a `RemoteEndpoint.Async` object. This object provides non-blocking methods for sending a message.

     To obtain a `RemoteEndpoint.Async` object, invoke the `Session.getAsyncRemote()` method.

3. Use the `RemoteEndpoint` object that you obtained in the previous step to send the message to the peer.

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

The following list shows some of the methods you can use to send a message to the peer:

- `void RemoteEndpoint.Basic.sendText(String text)`

  Send a text message to the peer. This method blocks until the whole message has been transmitted.

- `void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)`

  Send a binary message to the peer. This method blocks until the whole message has been transmitted.

- `void RemoteEndpoint.sendPing(ByteBuffer appData)`

  Send a ping frame to the peer.

- `void RemoteEndpoint.sendPong(ByteBuffer appData)`

  Send a pong frame to the peer.

Example 20-15 demonstrates how to use this procedure to reply to every incoming text message. For an example of how to send a message as the return value of a method, see Example 20-8.

Example 20-15 Sending a Message to a Single Peer of an Endpoint

This example replies to every incoming text message by sending the message back to the peer of this endpoint.

```
import java.io.IOException;

import jakarta.websocket.OnMessage;
import jakarta.websocket.Session;
...
   @OnMessage
   public void onMessage(Session session, String msg) {
      try {
         session.getBasicRemote().sendText(msg);
      } catch (IOException e) { ... }
   }
...
```

Sending a Message to All Peers of an Endpoint

Some WebSocket applications must send messages to all connected peers of the application's WebSocket endpoint, for example:

- A stock application must send stock prices to all connected clients.
- A chat application must send messages from one user to all other clients in the same chat room.
- An online auction application must send the latest bid to all bidders on an item.

However, each instance of an endpoint class is associated with one and only one connection and peer. Therefore, to send a message to all peers of an endpoint, you must iterate over the set of all open WebSocket sessions that represent connections to the same endpoint.

To send a message to all peers of an endpoint:

1. Obtain the set of all open WebSocket sessions that represent connections to the endpoint.

   Invoke the `getOpenSessions` method on the endpoint's `Session` object for this purpose.

2. Send the message to each open session that you obtained in the previous step.

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

a. Use the session to obtain a `RemoteEndpoint` object.

b. Use the `RemoteEndpoint` object to send the message.

See Sending a Message to a Single Peer of an Endpoint

This complete top-level endpoint demonstrates iterating over the same endpoint's open sessions and sending text to each peer. It intentionally omits a production send-coordination mechanism; read the warning immediately after the example.

Example 20-16 Sending a Message to All Peers of an Endpoint

This example forwards incoming text messages to all connected peers.

```
package examples.websocket;
import java.io.IOException;
import jakarta.websocket.OnMessage;
import jakarta.websocket.Session;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint("/echoall")
public class EchoAllEndpoint {
    @OnMessage
    public void messageReceived(Session sender, String message) {
        // Demonstrates peer iteration only. See the concurrency warning below.
        for (Session peer : sender.getOpenSessions()) {
            if (!peer.isOpen()) continue;
            try {
                peer.getBasicRemote().sendText(message);
            } catch (IOException | IllegalStateException failure) {
                System.getLogger(EchoAllEndpoint.class.getName()).log(
                        System.Logger.Level.WARNING, "Broadcast send failed", failure);
            }
        }
    }
}
```

Important

Concurrency and blocking: Callbacks for different connections can execute concurrently and attempt to send to the same receiving Session. This example does not coordinate those sends: BasicRemote.sendText can throw IllegalStateException when another send is already in progress. Catching and logging that exception does not retry the message or guarantee its delivery. BasicRemote.sendText is also blocking, so a slow peer can delay the callback and the remaining peers in this loop. For production broadcast, coordinate all writers through a bounded outbound queue per receiving Session, with one asynchronous send in flight at a time; advance the queue from send-completion callbacks. Configure a finite asynchronous send timeout and an explicit policy for send failures, queue overflow and slow clients, such as disconnecting a slow client. Keep queue-state locks brief and perform network sends outside those locks. Simply replacing getBasicRemote with getAsyncRemote does not supply bounded queuing or an application-level delivery policy.

The open-session set is a snapshot: a peer can close before a send, so both IOException and IllegalStateException are handled.

Ensuring Efficiency when Sending a Message to All Peers of an Endpoint

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

A blocking broadcast can be delayed by slow recipients. For high-volume workloads, consider asynchronous sends or bounded work submitted through managed concurrency services. Define a per-peer send policy, ordering requirements, queue limits, send timeouts and a slow-consumer policy. Handle asynchronous completion/failure and avoid an unbounded queue for a slow client. Additional concurrency can improve some workloads but can also add contention and resource pressure; measure under representative load. See Threading and Clustering Topics for managed-concurrency guidance.

Ensuring Thread Safety for WebSocket Endpoints

The container serializes its callbacks to an endpoint instance for each peer. This does not serialize callbacks across distinct peers sharing an instance, access to static/shared data, or tasks submitted by application code. Coordinate those accesses explicitly. Use the server's managed concurrency services when background work is needed, as described in Chapter 17, and coordinate concurrent sends to each peer. Do not infer a single thread for the entire endpoint class or application.

Encoding and Decoding a WebSocket Message

The Jakarta API for WebSocket provides support for converting between WebSocket messages and custom Java types by using encoders and decoders. This mechanism simplifies WebSocket applications because it decouples the business logic from the serialization and deserialization of objects.

An encoder takes a Java object and produces a representation that can be transmitted as a WebSocket text message or binary message. For example, encoders typically produce JavaScript Object Notation (JSON), Extensible Markup Language (XML), or binary representations. A decoder performs the reverse function: it reads a WebSocket message and creates a Java object.

> **Note**
>
> You may configure multiple decoder classes for a native WebSocket message type. Decoder.Text.willDecode or Decoder.Binary.willDecode identifies messages a decoder can process. A shared Java base type can be useful when one message handler must accept several application types; it is an application design choice, not a requirement to configure multiple decoders.

Encoding a Java Object as a WebSocket Message

You can have more than one encoder for text messages and more than one encoder for binary messages. Like endpoints, encoder instances are associated with one and only one WebSocket connection and peer. Therefore, only one thread is executing the code of an encoder instance at any given time.

To encode a Java object as a WebSocket message:

1. For each custom Java type that you want to send as a WebSocket message, implement the appropriate interface for the type of the WebSocket message:
   - For a text message, implement `jakarta.websocket.Encoder.Text<T>`.
   - For a binary message, implement `jakarta.websocket.Encoder.Binary<T>`.

   These interfaces specify the `encode` method.

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

2. Specify that your endpoint will use your encoder implementations.
   - For an annotated endpoint, add the names of your encoder implementations to the `encoders` optional element of the `ServerEndpoint` annotation.
   - For a programmatic endpoint, pass a list of the names of your encoder implementations as a parameter of the `encoders` method of a `jakarta.websocket.server.ServerEndpointConfig.Builder` object.
3. Use the `sendObject(Object data)` method of the `RemoteEndpoint.Basic` or `RemoteEndpoint.Async` interfaces to send your objects as messages.

   The complete encoders above implement the A:<text> and B:<text> demonstration format. MessageSender.sendBoth requires a connected Session configured with both encoders; its caller handles IOException and EncodeException.

The following examples show how to send the Java types `com.example.game.message.MessageA` and `com.example.game.message.MessageB` as text messages:

- Example 20-17
- Example 20-18
- Example 20-19

This compact format demonstrates codec behavior; the earlier JSON/XML discussion describes other valid encoding choices. Wire values are A:<text> and B:<text>. Place all model/codec classes in their declared package directories in the same WAR. Early and final EncEndpoint classes are alternatives; do not package both.

```
package com.example.game.message;
public abstract class Message {
    public abstract String getText();
}


package com.example.game.message;
public final class MessageA extends Message {
    private final String text;
    public MessageA(String text) { this.text =
java.util.Objects.requireNonNull(text); }
    @Override public String getText() { return text; }
}


package com.example.game.message;
public final class MessageB extends Message {
    private final String text;
    public MessageB(String text) { this.text =
java.util.Objects.requireNonNull(text); }
    @Override public String getText() { return text; }
}
```

Example 20-17 Implementing an Encoder Interface

This example implements the `Encoder.Text<MessageA>` interface.

```
package com.example.game.encoder;
import jakarta.websocket.Encoder;
```


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

```
import com.example.game.message.MessageA;
public class MessageATextEncoder implements Encoder.Text<MessageA> {
    @Override public String encode(MessageA message) { return "A:" + message.getText(); }
}


package com.example.game.encoder;
import jakarta.websocket.Encoder;
import com.example.game.message.MessageB;
public class MessageBTextEncoder implements Encoder.Text<MessageB> {
    @Override public String encode(MessageB message) { return "B:" +
message.getText(); }
}
```

The complete encoders above implement the A:<text> and B:<text> demonstration format. MessageSender.sendBoth requires a connected Session configured with both encoders; its caller handles IOException and EncodeException.

Example 20-18 Defining Encoders for an Annotated WebSocket Endpoint

This example defines the encoder classes `MessageATextEncoder.class` and `MessageBTextEncoder.class` for the WebSocket server endpoint `EncEndpoint`.

```
package com.example.game;
import jakarta.websocket.server.ServerEndpoint;
import com.example.game.encoder.MessageATextEncoder;
import com.example.game.encoder.MessageBTextEncoder;

@ServerEndpoint(value = "/myendpoint",
    encoders = {MessageATextEncoder.class, MessageBTextEncoder.class})
public class EncEndpoint {
}
```

Example 20-19 Sending Java Objects Encoded as WebSocket Messages

This example uses the `sendObject` method to send `MessageA` and `MessageB` objects as WebSocket messages.

```
package com.example.game;
import java.io.IOException;
import jakarta.websocket.EncodeException;
import jakarta.websocket.Session;
import com.example.game.message.MessageA;
import com.example.game.message.MessageB;
public final class MessageSender {
    private MessageSender() { }
    public static void sendBoth(Session session) throws IOException, EncodeException {
        session.getBasicRemote().sendObject(new MessageA("first"));
        session.getBasicRemote().sendObject(new MessageB("second"));
    }
}
```

Decoding a WebSocket Message as a Java Object

An endpoint may configure a list of decoder classes. This is separate from the restriction on native text, binary and pong message handlers. Like endpoints, decoder instances are associated with one and only one WebSocket connection and peer, so only one thread is executing the code of a decoder instance at any given time.

To decode a WebSocket message as a Java object:

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

1. Implement the appropriate interface for the type of the WebSocket message:
   - For a text message, implement `jakarta.websocket.Decoder.Text<T>`.
   - For a binary message, implement `jakarta.websocket.Decoder.Binary<T>`.

   These interfaces specify the `willDecode` and `decode` methods.
2. Specify that your endpoint will use your decoder implementations.
   - For an annotated endpoint, add the names of your decoder implementations to the `decoders` optional element of the `ServerEndpoint` annotation.
   - For a programmatic endpoint, pass a list of the names of your decoder implementations as a parameter of the `decoders` method of a `jakarta.websocket.server.ServerEndpointConfig.Builder` object.
3. Ensure that the method in your endpoint for handling a message received event takes your custom Java type as a parameter.

   See Handling Life Cycle Events for a WebSocket Connection.

   When the endpoint receives a message that can be decoded by one of the decoders you specified, the container calls the method that takes your custom Java type as a parameter if this method exists.

The following examples show how to decode WebSocket text messages as the Java types `com.example.game.message.MessageA` and `com.example.game.message.MessageB`:

- Example 20-20
- Example 20-21
- Example 20-22

These examples assume that the Java types `com.example.game.message.MessageA` and `com.example.game.message.MessageB` extend the `com.example.game.message.Message` class.

Example 20-20 Implementing a Decoder Interface

This example implements the `Decoder.Text<Message>` interface.

This example chooses one decoder for the common Message base type so a single handler can receive MessageA and MessageB.

```
package com.example.game.decoder;
import jakarta.websocket.DecodeException;
import jakarta.websocket.Decoder;
import com.example.game.message.Message;
import com.example.game.message.MessageA;
import com.example.game.message.MessageB;
public class MessageTextDecoder implements Decoder.Text<Message> {
    @Override
    public boolean willDecode(String text) {
        return text != null && (text.startsWith("A:") || text.startsWith("B:"));
    }
    @Override
    public Message decode(String text) throws DecodeException {
        if (!willDecode(text)) throw new DecodeException(text, "Expected A: or B:
prefix");
        return text.startsWith("A:") ? new MessageA(text.substring(2)) : new
MessageB(text.substring(2));
    }
}
```


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

Example 20-21 Defining a Decoder for an Annotated WebSocket Endpoint

This example defines the decoder class `MessageTextDecoder.class` for the WebSocket server endpoint `EncEndpoint`.

For completeness, this example also includes the definitions of the encoder classes `MessageATextEncoder.class` and `MessageBTextEncoder.class` from Example 20-18.

```
package com.example.game;
import jakarta.websocket.OnMessage;
import jakarta.websocket.server.ServerEndpoint;
import com.example.game.encoder.MessageATextEncoder;
import com.example.game.encoder.MessageBTextEncoder;
import com.example.game.decoder.MessageTextDecoder;
import com.example.game.message.Message;
import com.example.game.message.MessageA;
import com.example.game.message.MessageB;

@ServerEndpoint(value = "/myendpoint",
    encoders = {MessageATextEncoder.class, MessageBTextEncoder.class},
    decoders = {MessageTextDecoder.class})
public class EncEndpoint {
    @OnMessage
    public String message(Message message) {
        if (message instanceof MessageA) return "A received: " + message.getText();
        if (message instanceof MessageB) return "B received: " + message.getText();
        throw new IllegalArgumentException("Unexpected message type");
    }
}
```

This method-only extract belongs inside the final EncEndpoint class in Example 19-21.

Example 20-22 Receiving WebSocket Messages Encoded as Java Objects

This example defines the method `message` that receives `MessageA` objects and `MessageB` objects.

```
@OnMessage
    public String message(Message message) {
        if (message instanceof MessageA) return "A received: " + message.getText();
        if (message instanceof MessageB) return "B received: " + message.getText();
        throw new IllegalArgumentException("Unexpected message type");
    }
```

Specifying a Part of an Endpoint Deployment URI as an Application Parameter

The `ServerEndpoint` annotation enables you to use a level 1 URI template to specify parts of an endpoint deployment URI as application parameters. A URI template describes a range of URIs through variable expansion.

For more information about URI templates, see https://www.rfc-editor.org/rfc/rfc6570.

To specify a part of an endpoint deployment URI as an application parameter:

1. Set the `value` element of the `ServerEndpoint` annotation to the URI template that you want to use.

   In the URI template, enclose each variable for expansion in a pair of braces.

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

2. Declare each variable for expansion as a parameter in a method for handling one of the following types of event:
   - Connection opened
   - Connection closed
   - Message received
   - Error

   The type of the parameter can be `String`, a primitive type, or a boxed version of them.
3. Annotate the declaration of the parameter with the `jakarta.websocket.server.PathParam` annotation.
4. Set the value element of the `PathParam` annotation to the name of the variable.
5. Use the extracted parameter value in your application logic; the container performs URI matching and parameter binding.

Example 20-23 shows how to specify a part of an endpoint deployment URI as an application parameter.

Example 20-23 Specifying a Part of an Endpoint Deployment URI as an Application Parameter

This example specifies an endpoint deployment URI as a URI template that contains the variable `{room-name}`. The variable is expanded through the `roomName` parameter of the `open` method to determine which chat room the user wants to join.

```
package examples.websocket;
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.PathParam;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
    @OnOpen
    public void open(Session session, @PathParam("room-name") String roomName) {
        session.getUserProperties().put("roomName", roomName);
    }
}
```

The container extracts the {room-name} path segment and passes it as roomName.

If the endpoint is deployed inside a web application called `chatapp` at a local Jakarta EE server in port 8080, clients can connect to the endpoint using any of the following URIs:

```
ws://localhost:8080/chatapp/chatrooms/currentnews
ws://localhost:8080/chatapp/chatrooms/music
ws://localhost:8080/chatapp/chatrooms/cars
ws://localhost:8080/chatapp/chatrooms/technology
```

This example records the selected room name; it does not implement room authorization or a full chat service. /chatapp is the configured context root. Use wss for a remote secure deployment.

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

Maintaining Client State

Because the container creates an instance of the endpoint class for every connection, you can define and use instance variables to store client state information.

In addition, the `Session.getUserProperties` method provides a modifiable map to store user properties.

To store information common to all connected clients, you can use class (static) variables; however, you are responsible for ensuring thread-safe access to them.

Example 20-24 shows how to maintain client state.

Example 20-24 Maintaining Client State

This example replies to incoming text messages with the contents of the previous message from each client.

```
import java.io.IOException;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/delayedecho")
public class DelayedEchoEndpoint {
   @OnOpen
   public void open(Session session) {
      session.getUserProperties().put("previousMsg", " ");
   }
   @OnMessage
   public void message(Session session, String msg) {
      String prev = (String) session.getUserProperties()
                                    .get("previousMsg");
      session.getUserProperties().put("previousMsg", msg);
      try {
         session.getBasicRemote().sendText(prev);
      } catch (IOException e) { ... }
   }
}
```

Configuring a Server Endpoint Programmatically

The Jakarta API for WebSocket enables you to configure how the container creates server endpoint instances.

You can provide custom endpoint configuration logic for:

- Accessing the details of the handshake request for a WebSocket connection
- Performing custom checks on the `Origin` HTTP header
- Modifying the WebSocket handshake response
- Choosing a WebSocket subprotocol from those requested by the client
- Controlling the instantiation and initialization of endpoint instances
- Specifying the extensions that a server endpoint will support

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

To configure a server endpoint programmatically:

1. Extend the `jakarta.websocket.server.ServerEndpointConfig.Configurator` class.
2. Override the methods that perform the configuration operations for which you require custom logic, as shown in the following table.

Configuration Operation	Method to Override
Accessing the details of the handshake request for a WebSocket connection	`modifyHandshake`
Performing custom checks on the `Origin` HTTP header	`checkOrigin`
Modifying the WebSocket handshake response	`modifyHandshake`
Choosing a WebSocket subprotocol from those requested by the client	`getNegotiatedSubprotocol`
Controlling the instantiation and initialization of endpoint instances	`getEndpointInstance`
Specifying the extensions that a server endpoint will support	`getNegotiatedExtensions`

3. In the server endpoint class, set the `configurator` element of the `ServerEndpoint` annotation to the configurator class.

The following examples show how to configure a server endpoint programmatically:

- Example 20-25
- Example 20-26

modifyHandshake copies a snapshot of the request headers into this connection's configuration properties; onOpen uses the copied map. Do not retain a servlet-backed handshake request for later processing. The example counts header names and does not log or expose potentially sensitive header values.

Example 20-25 Extending the ServerEndpointConfig.Configurator Class

This example copies request headers during the handshake and uses that snapshot when the connection opens.

```
package examples.websocket;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import jakarta.websocket.HandshakeResponse;
import jakarta.websocket.server.HandshakeRequest;
import jakarta.websocket.server.ServerEndpointConfig;

public class CustomConfigurator extends ServerEndpointConfig.Configurator {
    @Override
    public void modifyHandshake(ServerEndpointConfig config,
            HandshakeRequest request, HandshakeResponse response) {
        Map<String, List<String>> headers = new LinkedHashMap<>();
        request.getHeaders().forEach((name, values) -> headers.put(name,
List.copyOf(values)));
        config.getUserProperties().put("requestHeaders", Map.copyOf(headers));
    }
}
```


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

Example 20-26 Specifying a Custom Configurator for a Server Endpoint Class

This example specifies the custom configurator class `CustomConfigurator.class` for the server endpoint class `MyEndpoint`.

This example copies request headers during the handshake and uses that snapshot when the connection opens.

```
package examples.websocket;
import java.util.List;
import java.util.Map;
import jakarta.websocket.EndpointConfig;
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint(value = "/configured", configurator = CustomConfigurator.class)
public class MyEndpoint {
    @OnOpen
    @SuppressWarnings("unchecked")
    public void open(Session session, EndpointConfig config) {
        Map<String, List<String>> headers = (Map<String, List<String>>)
                config.getUserProperties().get("requestHeaders");
        session.getUserProperties().put("headerCount", headers.size());
    }
}
```

Building Applications that Use the Jakarta API for WebSocket

For a full WebLogic installation, compile with WL_HOME/server/lib/api.jar, where WL_HOME is the installation's wlserver directory.

For server application compilation, declare the Jakarta WebSocket 2.2 API with provided scope, because WebLogic supplies the implementation. The common/client API is in jakarta.websocket-client-api and the server API is in jakarta.websocket-api. These API artifacts alone do not provide a standalone client implementation. See Using the WebLogic Maven Plug-In.

```
<dependencies>
  <dependency>
    <groupId>jakarta.websocket</groupId>
    <artifactId>jakarta.websocket-api</artifactId>
    <version>2.2.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>jakarta.websocket</groupId>
    <artifactId>jakarta.websocket-client-api</artifactId>
    <version>2.2.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

A standalone Java client also needs a compatible implementation and transport. Use the matching WebLogic client runtime/classpath; keep a competing implementation out of the WAR.

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

Deploying a WebSocket Application

In WebLogic Server, you deploy a WebSocket application as part of a standard Jakarta EE Web application archive (WAR), either as a standalone Web application or a WAR module within an enterprise application.

Annotated server endpoints can be discovered automatically in a WAR, subject to endpoint-selection configuration. Programmatic server endpoints require registration, for example through ServerApplicationConfig as shown in Example 19-6. Complete registration during application initialization.

However, you can optionally set the context initialization properties that are listed in Table 20-3. To indicate that these properties are specific to WebLogic Server and not part of the Jakarta WebSocket specification, their fully qualified names contain the prefix `weblogic.websocket`.

Table 20-3 Context Initialization Properties for a WebSocket Application

Property	Type	Description
`weblogic.websocket.tyrus.incoming-buffer-size`	`Integer`	Maximum underlying receive-buffer capacity in bytes. This implementation buffer setting is distinct from whole-message limits and buffering configured through @OnMessage.maxMessageSize and Session/WebSocketContainer message-buffer APIs. The default capacity is 4194315 bytes. This parameter affects the following server sessions and client sessions: • **All** server sessions in the same application • **Only** client sessions that are connected with the server-instantiated `jakarta.websocket.server.ServerContainer` object in the application You can override this setting for clients sessions by setting a property of the same name for a client endpoint. For more information, see Configuring a WebSocket Client Endpoint Programmatically.
`weblogic.websocket.tyrus.session-max-idle-timeout`	long, milliseconds	The maximum period in milliseconds after which an idle connection times out. The default value is 30000, which corresponds to 30 seconds.
`weblogic.websocket.tyrus.cluster`	`String`	Optional cluster-aware messaging uses Coherence; capacity and broadcast performance depend on topology and workload. Clustering is disabled by default. To enable clustering set the value to *true*.

Example 20-27 shows how to set context initialization properties for a WebSocket application.

Example 20-27 Setting Context Initialization Properties for a WebSocket Application

This example sets context initialization parameters for a WebSocket application as follows:

- The maximum underlying buffer size for receiving messages is set to 16777227 bytes.
- The maximum period after which an idle connection times out is set to 60,000 milliseconds, which corresponds to 1 minute.
- Enable WebSocket cluster using managed Coherence server to establish communication among all members.

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

> **Note**
>
> Clustering requires a managed Coherence server with local storage enabled. See, Configure Coherence Cluster Member Storage Settings in *Administering Clusters for Oracle WebLogic Server*.

```
<?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">
  <context-param>
    <param-name>weblogic.websocket.tyrus.incoming-buffer-size</param-name>
    <param-value>16777227</param-value>
  </context-param>
  <context-param>
    <param-name>weblogic.websocket.tyrus.session-max-idle-timeout</param-name>
    <param-value>60000</param-value>
  </context-param>
  <context-param>
    <param-name>weblogic.websocket.tyrus.cluster</param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

These are separate minimal web.xml examples. If using both features, merge their context-param elements into one existing web-app root, retaining the application's other configuration; do not paste two roots or discard existing security settings.

Monitoring WebSocket Applications

You can monitor message statistics and runtime properties for WebSocket applications and endpoints. Endpoint-level monitoring collects information per individual endpoint, while application-level monitoring aggregates information from all endpoints deploying in the given application.

WebSocket Monitoring Properties

The following table details the types of properties monitored at runtime and whether monitoring occurs at the application or endpoint level. For message-related properties, WebLogic Server uses bytes for message size and distinguishes three types of messages: text, binary, and control.

Property	Description	Monitoring Level
Open session count	The number of current open sessions for the WebSocket application or endpoint.	application, endpoint
Maximum open sessions count	The highest number of open sessions for the WebSocket application or endpoint since server startup.	application, endpoint
Error counts	The list of errors with the number of times each error has occurred. Errors are represented by throwable class names.	application, endpoint

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

Property	Description	Monitoring Level
Sent messages count	The number of sent messages for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Received messages count	The number of received messages for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Sent messages count per second	The number of sent messages per second for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Received messages count per second	The number of received messages per second for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Minimum sent message size	The smallest sent message size for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Minimum received message size	The smallest received message size for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Maximum sent message size	The largest sent message size for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint

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

Property	Description	Monitoring Level
Maximum received message size	The largest received message size for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Average sent message size	The average sent message size for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Average received message size	The average received message size for the WebSocket application or endpoint since monitoring began. Statistics are provided per individual message type (text, binary, and control) and as a total count.	application, endpoint
Endpoint path	The path on which the endpoint is registered, relative to the application context root.	endpoint only
Endpoint class name	The name of the endpoint class.	endpoint only

To access monitored metrics for WebSocket applications and endpoints at runtime, use the following MBeans:

- `WebAppComponentRuntimeMBean`
- `WebsocketApplicationRuntimeMBean`
- `WebsocketBaseRuntimeMBean`
- `WebsocketEndpointRuntimeMBean`
- `WebsocketMessageStatisticsRuntimeMBean`

Using WebSockets with Proxy Servers

Clients accessing WebSocket applications must either connect directly to the WebLogic Server instance or through a Web proxy server that supports the WebSocket protocol.

The following proxy servers support the WebSocket protocol:

- Oracle HTTP Server
- Apache HTTP Server when used with the Oracle WebLogic Server Proxy Plug-In

For information about the specific versions of Apache HTTP Server supported for use with the Oracle WebLogic Server Proxy Plug-In, see the Oracle Fusion Middleware Supported System Configurations page on the Oracle Technology Network.

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

Writing a WebSocket Client

A WebSocket client application is typically a browser-based client. The Jakarta API for WebSocket can also be used to write a Jakarta WebSocket client.

Writing a Browser-Based WebSocket Client

A browser-based WebSocket client application is typically a composite of HTML5 technologies, including HTML markup, CSS3, and JavaScript that makes use of the WebSocket JavaScript API. For more information about HTML5, see http://www.w3.org/TR/html5/.

Most browsers support the W3C WebSocket API that can be used to create and work with the WebSocket protocol. For information about the W3C WebSocket API, see: http://www.w3.org/TR/websockets/.

If the WebSocket protocol is not guaranteed to be supported in the runtime environment, use the JavaScript API for WebSocket fallback in your browser-based client. This API provides an implementation of the standard W3C WebSocket API. The API also provides a mechanism for using an alternative transport for WebSocket messaging when the WebSocket protocol is not supported. For more information, see Enabling Protocol Fallback for WebSocket Messaging.

The following steps show an example of the execution flow on a client that is sending messages to a WebLogic Server instance using the WebSockets Protocol.

1. The client opens a WebSocket connection to the server hosting the WebSocket endpoint, using the `ws://` or `wss://` protocol prefix. For more information, see Establishing Secure WebSocket Connections.

```
var url = ((window.location.protocol == "https:") ? "wss:" : "ws:")
+ "//" + window.location.host
+ "/websocket-helloworld-wls/helloworld_delay.ws";

var ws = new WebSocket(url);
```

2. The client registers listeners with the WebSocket object to respond to events, such as opening, closing, and receiving messages. Based on the event and the information received, the client performs the appropriate action.

```
ws.onopen = function(event) {
    document.getElementById("status").innerHTML = "OPEN"
}

ws.onmessage = function(event) {
    msg = event.data
      document.getElementById("short_msg").innerHTML =
      event.data;
}
```

3. The client sends messages to the server over the WebSocket object as needed by the application.

```
function sendMsg() {
    // Check if connection is open before sending
    if(ws == null || ws.readyState != 1) {
        document.getElementById("reason").innerHTML
        = "Not connected can't send msg"
    } else {
        ws.send(document.getElementById("name").value);
    }
}
```


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

```
<input id="send_button" class="button" type="button" value="send"
onclick="sendMsg()"/>
```

Writing a Java WebSocket Client

The `jakarta.websocket` package contains annotations, classes, interfaces, and exceptions that are common to client and server endpoints. Use the APIs in this package for writing a Java WebSocket client in the same way as for writing a server. Additional programming tasks that are specific to writing a client are described in the subsections that follow.

Configuring a WebSocket Client Endpoint Programmatically

WebLogic Server provides properties for configuring how the container creates client endpoint instances. To indicate that these properties are specific to WebLogic Server and not part of the Jakarta WebSocket specification, their fully qualified names contain the prefix `weblogic.websocket`.

WebLogic Server provides properties for the following:

- **HTTP proxy configuration.** WebLogic Server supports client connections to a remote server WebSocket endpoint through an HTTP proxy as defined in the WebSocket Protocol (RFC 6455).

  Properties for HTTP proxy configuration are listed in Table 20-4.
- **Secure Sockets Layer (SSL) configuration.** WebLogic Server supports client connections to a remote server WebSocket endpoint over SSL with wss scheme.

  Properties for SSL configuration are listed in Table 20-5.
- **Buffer size for incoming messages.** WebLogic Server supports limiting the size of incoming messages for WebSocket client endpoints.

  Properties for buffer size configuration are described in Table 20-6.

Table 20-4 HTTP Proxy Configuration Properties for a Java WebSocket Client

Property	Type	Description
`weblogic.websocket.client.PROXY_HOST`	`String`	Required when selecting this proxy. Name of the HTTP proxy host for the Java WebSocket client.
`weblogic.websocket.client.PROXY_PORT`	`Integer`	Optional. The port number for connections to the HTTP proxy host. If you specify an HTTP proxy host without the port number, the port number defaults to 80.
`weblogic.websocket.client.PROXY_USERNAME`	`String`	Optional. The user name for logging in to the proxy host.
`weblogic.websocket.client.PROXY_PASSWORD`	`String`	Optional. Password for logging in to the proxy host.

Table 20-5 TLS Configuration Properties

Property	Type	Description
`weblogic.websocket.client.SSL_PROTOCOLS`	`String`	Optional comma-separated TLS protocol versions enabled for this client connection, restricted to versions supported by the configured runtime/security policy.

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

Table 20-5 (Cont.) TLS Configuration Properties

Property	Type	Description
`weblogic.websocket.client.SSL_TRUSTSTORE`	`String`	Optional path to the client truststore containing trusted certificates used to validate the server's certificate chain.
`weblogic.websocket.client.SSL_TRUSTSTORE_PWD`	`String`	Optional. Password for the client truststore.

These Java implementation properties do not configure a browser's proxy or trust settings. Keep certificate-chain and hostname verification enabled.

Table 20-6 Buffer-Size Configuration Properties for a Java WebSocket Client

Property	Type	Description
`weblogic.websocket.tyrus.incoming-buffer-size`	`Integer`	Maximum underlying receive-buffer capacity in bytes. This implementation buffer setting is distinct from whole-message limits and buffering configured through @OnMessage.maxMessageSize and Session/WebSocketContainer message-buffer APIs. The default capacity is 4194315 bytes. If set, this property overrides the value of the context initialization property of the same name that is described in Table 20-3.

> **Note**
>
> Configure a client endpoint before connecting the client to its server endpoint.

To configure a WebSocket client endpoint programmatically:

1. Obtain a `jakarta.websocket.ClientEndpointConfig` object.
   a. Invoke the `jakarta.websocket.ClientEndpointConfig.Builder.create` static method to obtain an instance of the `ClientEndpointConfig.Builder` class.
   b. Invoke the `build` method on the `ClientEndpointConfig.Builder` object that you obtained in the previous step.
2. Set each configuration property that you want to change to its new value.
   a. Invoke the `getUserProperties` method on the `ClientEndpointConfig` object that you obtained in the previous step to obtain a modifiable `java.util.Map` object that contains the user properties.
   b. Invoke the `put` method on the `Map` object that you obtained in the previous step.

      In the invocation of the `put` method, provide the property name and its new value as parameters to the method.

Example 20-28 shows how to configure a WebSocket client endpoint programmatically.

This method builds configuration only. Supply the actual proxy host/Integer port, truststore path and password through your application's protected configuration; do not commit a real password or log this map. Pass the returned ClientEndpointConfig as the cec argument of a

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

programmatic-endpoint connectToServer overload shown in the next table. An annotated endpoint's connectToServer(Class, URI) overload does not take this object. A standalone client needs the release-matched WebLogic/Tyrus client implementation; these are implementation-specific keys.

Example 20-28 Configuring a WebSocket Client Endpoint Programmatically

The method receives the proxy host and Integer port plus the protected truststore path/password. It selects receive-buffer capacity 16777227 bytes.

```
package examples.websocket;
import jakarta.websocket.ClientEndpointConfig;

public final class ClientSettings {
    private ClientSettings() { }
    public static ClientEndpointConfig create(String proxyHost, int proxyPort,
            String trustStore, String trustStorePassword) {
        ClientEndpointConfig config = ClientEndpointConfig.Builder.create().build();
        config.getUserProperties().put("weblogic.websocket.client.PROXY_HOST",
proxyHost);
        config.getUserProperties().put("weblogic.websocket.client.PROXY_PORT",
proxyPort);
        config.getUserProperties().put("weblogic.websocket.client.SSL_TRUSTSTORE",
trustStore);
        config.getUserProperties().put("weblogic.websocket.client.SSL_TRUSTSTORE_PWD",
trustStorePassword);
        config.getUserProperties().put("weblogic.websocket.tyrus.incoming-buffer-size",
16 * 1024 * 1024 + 11);
        return config;
    }
}
```

Connecting a Java WebSocket Client to a Server Endpoint

Choose an annotated or programmatic overload and provide either an endpoint class for container creation or an application-created instance. In a Jakarta EE-managed environment, class-based creation permits the container to perform supported injection. A Java SE client does not obtain CDI or other Jakarta EE services merely by passing a Class object.

1. Invoke the `jakarta.websocket.ContainerProvider.getWebSocketContainer()` static method to obtain the client's `jakarta.websocket.WebSocketContainer` instance.
2. Invoke the overloaded `connectToServer` method on the `WebSocketContainer` object that you obtained in the previous step.

   The variant of the method to invoke depends on whether the endpoint is an annotated endpoint or a programmatic endpoint and whether support for Jakarta EE services such as dependency injection are required.

Endpoint Type	Container creation needed for supported Jakarta EE services (managed environment only)	Variant of the connectToServer Method
Annotated	Not required	connectToServer(Object annotatedEndpointInstance, URI path)

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

Endpoint Type	Container creation needed for supported Jakarta EE services (managed environment only)	Variant of the connectToServer Method
Annotated	Required	connectToServer(Class<?> annotatedEndpointClass, URI path)
Programmatic	Not required	connectToServer(Endpoint endpointInstance, ClientEndpointConfig cec, URI path)
Programmatic	Required	connectToServer(Class<? extends Endpoint> endpointClass, ClientEndpointConfig cec, URI path)

In the invocation of the `connectToServer` method, provide the following information as parameters to the method:

- The **client** WebSocket endpoint
- The complete path to the **server** WebSocket endpoint

If the **client** endpoint is a programmatic endpoint, you must also provide configuration information for the endpoint.

Example 20-4 shows how to connect a Java WebSocket client to a server endpoint.

Compile/package both classes in examples.websocket. Run with the full endpoint URI as the sole argument. This example demonstrates opening and closing a connection only; put actual client work inside the try block before it exits. try-with-resources closes the Session on success or failure.

Example 20-29 Connecting a Java WebSocket Client to a Server Endpoint

This example connects the Java WebSocket client `ClientExample` to the WebSocket server endpoint at `ws://example.com:80/echoserver/echo`. The WebSocket client endpoint is represented by the class `ExampleEndpoint`. The declaration of the `ExampleEndpoint` class is shown in Example 20-4.

```
package examples.websocket;
import jakarta.websocket.ClientEndpoint;
@ClientEndpoint
public class ExampleEndpoint {
}
```

Use the matching WebLogic client runtime. The host, port and context root are illustrative; configure TLS trust for the actual endpoint. On Windows use a semicolon classpath separator.

```
java -cp ".:$WL_HOME/server/lib/weblogic.jar"
examples.websocket.ClientExample wss://server.example.com:7002/echoapp/echo
```

Setting the Maximum Number of Threads for Dispatching Messages from a WebSocket Client

By default, the maximum number of threads for dispatching messages from a WebSocket client depends on how many processors are available:

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

- If 20 or fewer processors are available, the maximum number of threads is 20.
- If more than 20 processors are available, the maximum number of threads is equal to the number of available processors.

To set the maximum number of threads for dispatching messages from a WebSocket client:

- In the `java` command to launch your client application, set the system property `weblogic.websocket.client.max-aio-threads` to the number that you require.

Example 20-30 shows how to set the maximum number of threads for dispatching messages from a WebSocket client.

For this client runtime, specify an integer maximum of at least 3; this controls the client's asynchronous I/O pool, not the WebLogic Server's managed executor pool. The value 50 is illustrative. Replace endpoint host/port/context root and configure TLS trust.

Example 20-30 Setting the Maximum Number of Threads for Dispatching Messages from a WebSocket Client

This example sets the maximum number of threads for dispatching messages from the WebSocket client `ClientExample` to 50.

```
java -Dweblogic.websocket.client.max-aio-threads=50 -cp ".:$WL_HOME/server/lib/
weblogic.jar" examples.websocket.ClientExample wss://server.example.com:7002/echoapp/echo
```

Securing a WebSocket Application

In WebLogic Server, you deploy a WebSocket application as a Web application archive (WAR), either as a standalone Web application or a WAR module within an enterprise application. Therefore, many security practices that you apply to securing Web applications can apply to WebSocket applications.

For information about Web application security, see Developing Secure Web Applications in *Developing Applications with the WebLogic Security Service*.

Applying Verified-Origin Policies

Modern browsers use same-origin policies to prevent scripts that are running on Web pages loaded from one origin from interacting with resources from a different origin. The WebSocket Protocol (RFC 6455) uses a verified-origin policy that enables the server to decide whether or not to consent to a cross-origin connection.

When a script sends an opening handshake request to a WebSocket application, an `Origin` HTTP header is sent with the WebSocket handshake request. If the application does not verify the `Origin`, then it accepts connections from any origin. If the application is configured not to accept connections from origins other than the expected origin, then the WebSocket application can reject the connection.

You can ensure that the WebSocket application verifies the `Origin` by extending the `jakarta.websocket.server.ServerEndpointConfig.Configurator` class.

The following code example demonstrates applying a verified-origin policy:

Replace the illustrative HTTPS page origin with the exact allowed origin for your application, including a non-default port when applicable. This strict policy rejects missing or different Origin values. Register the configurator on each server endpoint to which the policy applies. Test the policy with the externally visible origin when a proxy is involved.

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

```
package examples.websocket;
import jakarta.websocket.server.ServerEndpointConfig;

public class MyConfigurator extends ServerEndpointConfig.Configurator {
    private static final String ALLOWED_ORIGIN = "https://app.example.com";
    @Override
    public boolean checkOrigin(String origin) {
        return ALLOWED_ORIGIN.equals(origin);
    }
}
```

```
package examples.websocket;
import jakarta.websocket.OnMessage;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint(value = "/demo", configurator = MyConfigurator.class)
public class OriginEndpoint {
    @OnMessage
    public String echo(String message) { return message; }
}
```

For more information, see Configuring a Server Endpoint Programmatically.

> **Note**
>
> Browser clients send an Origin header during the opening handshake. Other clients can omit or forge it; Origin is not proof of identity. Decide explicitly how nonbrowser clients are allowed and authenticate/authorize them through the application's normal security controls.

Authenticating and Authorizing WebSocket Clients

The WebSocket Protocol (RFC 6455) does not specify an authentication method for WebSocket clients during the handshake process. You can use standard Web container authentication and authorization functionality to prevent unauthorized clients from opening WebSocket connections on the server.

All configurations of the `<auth-method>` element that are supported for Web applications can also be used for WebSocket applications. These authentication types include BASIC, FORM, CLIENT-CERT, and so on. See Developing Secure Web Applications in *Developing Applications with the WebLogic Security Service*.

Configure web.xml security-constraint for the endpoint path. The protected opening HTTP request must satisfy the configured web authentication and role constraints before it can upgrade. With FORM authentication, log in through the web application's HTTPS flow first, then open the WebSocket using the authenticated session cookie. Do not expect a WebSocket API connection attempt to display a login form or complete an interactive redirect flow. Configure nonbrowser client credentials for the selected authentication mechanism explicitly. See web.xml Deployment Descriptor Elements in Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server.

The following code example demonstrates securing the path to the endpoint within your application, where the path is `/demo`:

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

Provide the referenced login/error pages and security-realm-to-role mapping for role user before deploying this /demo example.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secured WebSocket Endpoint</web-resource-name>
    <url-pattern>/demo</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/error.jsp</form-error-page>
  </form-login-config>
</login-config>
<security-role>
  <role-name>user</role-name>
</security-role>
```

Authorizing WebSocket Clients

Server authorization determines whether the authenticated user may open the endpoint. Use the web.xml security constraint and role mapping shown in Authenticating and Authorizing WebSocket Clients; the client-side ClientEndpointConfig.Configurator does not grant server access. Endpoint code can obtain the authenticated identity using Session.getUserPrincipal(). Keep application-specific authorization for messages and room/resource access separate from this opening-handshake constraint. Origin validation is an additional browser-origin policy, not user authentication; see Applying Verified-Origin Policies.Authenticating and Authorizing WebSocket ClientsApplying Verified-Origin Policies

Establishing Secure WebSocket Connections

To establish a WebSocket connection, the client sends a handshake request to the server. When using the `ws` scheme to open the WebSocket connection, the handshake request is a plain HTTP request; the data being transferred over the established WebSocket connection is not encrypted.

To establish a secure WebSocket connection and prevent data from being intercepted, WebSocket applications should use the `wss` scheme. The `wss` scheme ensures that clients send handshake requests as HTTPS requests, encrypting transferred data by TLS/SSL.

You can configure a WebSocket application to accept only HTTPS handshake requests, where all WebSocket connections must be encrypted and unencrypted WebSocket handshake requests are rejected. Specify the `<user-data-constraint>` element in the `web.xml` deployment descriptor file of the WebSocket application. For more information about the `<user-data-constraint>` element, see web.xml Deployment Descriptor Elements in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

The following code example demonstrates configuring the `<user-data-constraint>` element:

```
<security-constraint>
    <web-resource-collection>
      <web-resource-name>Secured WebSocket Endpoint</web-resource-name>
      <url-pattern>/demo</url-pattern>
```


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

```
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Avoiding Mixed Content

Serve the page over HTTPS and connect with wss. Opening ws from an HTTPS page can be blocked as mixed content and leaves permitted unencrypted traffic exposed to observation or alteration. Do not downgrade to ws to work around a TLS failure. Protect session cookies with the application's appropriate Secure, HttpOnly and SameSite settings; do not rely on mixed-content blocking as the only protection.

WebSockets Standard

Mixed Content

Specifying Limits for a WebSocket Connection

By specifying limits for a WebSocket connection, you can help limit consumption of server resources, such as memory, sockets, and so forth.

You can specify the following limits for a WebSocket connection:

- For an annotated whole-message handler, set @OnMessage(maxMessageSize = 4096) as appropriate; it is not a connection-count limit. Configure the relevant Session/container message-buffer size and idle timeout for your application as well.
- **Idle timeout value.** To set the idle timeout value for a WebSocket connection, invoke one of the following methods:
  - For an individual connection, invoke the `setMaxIdleTimeout` method of the `Session` object.
  - For the entire container, invoke the `setDefaultMaxSessionIdleTimeout` method of a `WebSocketContainer` object.

Enabling Protocol Fallback for WebSocket Messaging

Protocol fallback provides a mechanism for using an alternative transport for WebSocket messaging when the WebSocket protocol is not supported. Typically the WebSocket protocol is not supported either because the WebSocket object is not available or because WebSocket frames are blocked by a firewall. In this release, the only supported alternative transport is HTTP Long Polling.

Protocol fallback enables you to rely on standard programming APIs to perform WebSocket messaging regardless of whether or not the runtime environment supports the WebSocket protocol.

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

> **Note**
>
> To support WebSocket fallback, the server must use the Jakarta WebSocket, not the proprietary WebLogic Server WebSocket API.

Using the JavaScript API for WebSocket Fallback in Client Applications

The JavaScript API for WebSocket fallback provides an implementation of the standard W3C WebSocket API and additional APIs to facilitate WebSocket fallback. For information about the JavaScript API for WebSocket fallback, see *JavaScript API Reference for WebSocket Fallback*. For information about the W3C WebSocket API, see: http://www.w3.org/TR/websockets/.

When you use the standard W3C WebSocket JavaScript API, code your application without regard to whether the WebSocket protocol is supported.

Configuring WebSocket Fallback

WebLogic Server provides properties for configuring WebSocket fallback as listed in Table 20-7.

Table 20-7 WebSocket Fallback Configuration Properties

Property	Type	Default	Description
`baseUrl`	string	.	The location of the `scripts` directory, relative to the HTML context of the page. The structure of the `scripts` directory **must** be preserved. The `scripts` directory can be moved to wherever it can be reached, but its content must not change after it was created.
`debug`	integer	0	The debug level.
`ENCODE_FOR_IE_BELOW`	integer	10	The version of the Internet Explorer browser below which Base16 encoding is to be used for framed data.
`ENFORCE_ENCODING`	Boolean	false	Whether Base16 encoding is to be used.
`NB_TRY_FOR_EACH_TRANSPORT`	integer	2	The maximum number of consecutive retries to establish a connection on a given transport.
`PING_INTERVAL`	integer	25000	Interval in milliseconds between consecutive pings to the server.
`SERVER_PING_ENABLED`	Boolean	true	Whether pings from the client to the server are enabled.
`transport`	string	none	optional string; omit for automatic transport selection; supported explicit strings are WebSocket and XMLHttpRequest.
`TRY_AGAIN_INTERVAL`	integer	10000	The number of milliseconds (10 seconds by default) after which an unsuccessful connection attempt is repeated with the same transport. The retry count for the transport is **not** incremented. If the attempt fails within this number of milliseconds, the retry count is incremented by 1.
`WEBSOCKET_CREATION_TIMEOUT`	integer	1000	The number of milliseconds after which creation of a WebSocket connection is considered to have failed.

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

These client-library retry and heartbeat settings are distinct from the server idle timeout.

If the WebSocket protocol is available, WebLogic Server uses that protocol even if protocol fallback is enabled. WebLogtic Server uses the value of the `TRY_AGAIN_INTERVAL` property and the `NB_TRY_FOR_EACH_TRANSPORT` property as follows to determine whether the WebSocket protocol is available if a connection attempt fails:

- If the connection is not established within `TRY_AGAIN_INTERVAL` milliseconds, the attempt is repeated with same transport. The retry count for this transport is **not** incremented.
- If the attempt fails within `TRY_AGAIN_INTERVAL` milliseconds, the retry count is incremented by 1.
- If the retry count reaches the value of `NB_TRY_FOR_EACH_TRANSPORT`, the next transport is tried.
- If the retry count for the last transport reaches the value of `NB_TRY_FOR_EACH_TRANSPORT`, the connection is closed, that is, the `onclose` function is called on the client.

To configure WebSocket fallback:

1. Construct a JavaScript object object in which you set the configuration properties that you require.

   For details about these properties, see Table 20-7.

2. Pass the object as a parameter to one of the following functions:

   - If the fallback mechanism cannot be guaranteed to be present, pass the object as the parameter to the `OraSocket.configure` function **before** constructing the WebSocket object.

     Check whether OraSocket is available before calling configure, as shown below.

   - Otherwise, pass the object as the second, optional parameter of the WebSocket object's constructor.

Example 20-31 shows how to configure WebSocket fallback.

Run this before constructing the socket. When OraSocket is present it enforces XMLHttpRequest fallback for this development demonstration, enables verbose debugging and disables the library heartbeat. If the library is absent, these fallback options are not applied; the native WebSocket API remains responsible for any connection. Do not pass this object as the native constructor’s second argument, which is reserved for subprotocols. The object argument is an OraSocket extension; prefer the explicit configure call to keep that distinction clear. Use debug level 10 only for local diagnostics. The fallback library must be loaded for these features.

Example 20-31 Configuring WebSocket Fallback

This example enforces the `XMLHttpRequest` transport, sets the debug level to 10, and disables pings from the client to the server.

```
if (typeof OraSocket !== "undefined") {
    OraSocket.configure({transport: "XMLHttpRequest", debug: 10, SERVER_PING_ENABLED:
false});
}
```

Creating a WebSocket Object

A WebSocket object represents a WebSocket connection from the client to a remote host.

To create a WebSocket object, invoke the `WebSocket` constructor, passing the following information as parameters:

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

- The URL to which the client should connect
- Optionally, a JSON object that contains configuration settings for WebSocket fallback

  For more information about the JSON object, see Configuring WebSocket Fallback.

Example 20-32 shows how to create a WebSocket object.

Load the optional fallback library and apply its configuration first. Serve this page from the same origin as the endpoint; /websocket-101/ws-101-app is the example endpoint path and must match the deployed context root/path. location.host already retains an explicit port and brackets an IPv6 address when needed. The three DOM elements must exist before running this script. If socket construction fails, the fragment reports and rethrows the error; do not install callbacks on an undefined socket.

```
<p id="status"></p>
<p id="prompt"></p>
<div id="content"></div>
```

Example 20-32 Creating a WebSocket Object

This example creates the WebSocket Object `ws`. The example uses standard JavaScript functions to determine the URL to which the client should connect from the URL of the document that contains this code.

```
var ws;
var statusFld = document.getElementById("status");
var promptFld = document.getElementById("prompt");
var contentFld = document.getElementById("content");
var wsURI = (location.protocol === "https:" ? "wss:" : "ws:")
    + "//" + location.host + "/websocket-101/ws-101-app";
try {
    ws = new WebSocket(wsURI);
} catch (error) {
    statusFld.textContent = "Unable to create WebSocket: " + String(error);
    throw error;
}
```

Handling Life Cycle Events for a JavaScript WebSocket Client

Handling lifecycle events for a JavaScript WebSocket client involves writing the `WebSocket` object's callback functions as listed in Table 20-8. The table also provides a cross-reference to an example that shows how to handle each type of event.

Table 20-8 Callback Functions for Handling Life Cycle Events

Event	Callback Function	Example
Connection opened	`onopen`	Example 20-33
Message received	`onmessage`	Example 20-34
Error	`onerror`	Example 20-35
Connection closed	`onclose`	Example 20-36

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

> **Note**
>
> The creation of the `ws` `WebSocket` object in the examples is shown in Example 20-32.

These examples use ws/statusFld/promptFld/contentFld initialized by Example 19-32. Messages use this illustrative application payload: {"type":"message","appdata":{"text":"hello"}}. Change the handler to your application protocol when needed. Malformed or unexpected payloads are ignored in this small display example. Received text is inserted as a text node and never interpreted as markup. Browser error events can omit detailed failure reasons; inspect application/server diagnostics for those. Define an application retention limit for a long-running message log.

Example 20-33 Handling a Connection Opened Event for a JavaScript WebSocket Client

This example uses standard JavaScript functions to display the current date and time followed by the message `Connection opened` when a connection is opened.

```
ws.onopen = function () {
    promptFld.textContent = "Message:";
    statusFld.textContent = new Date().toISOString() + ": Connection opened";
};
```

Example 20-34 Handling a Message Received Event for a JavaScript WebSocket Client

This example uses standard JavaScript functions to display the current time followed by the content of the message when a message is received.

```
ws.onmessage = function (event) {
    if (typeof event.data !== "string") return;
    var message;
    try { message = JSON.parse(event.data); } catch (error) { return; }
    if (!message || message.type !== "message" || !message.appdata
            || typeof message.appdata.text !== "string") return;
    var line = document.createElement("div");
    line.textContent = new Date().toISOString() + ": " + message.appdata.text;
    contentFld.appendChild(line);
    contentFld.scrollTop = contentFld.scrollHeight;
};
```

Example 20-35 Handling an Error Event for a JavaScript WebSocket Client

This example uses standard JavaScript functions to display the current date and time followed by an error message when an error occurs.

```
ws.onerror = function () {
    statusFld.textContent = new Date().toISOString() + ": WebSocket error; inspect
application diagnostics";
};
```

Example 20-36 Handling a Connection Closed Event for a JavaScript WebSocket Client

This example uses standard JavaScript functions to display the current date and time followed by the message `Connection closed` when a connection is closed.

```
ws.onclose = function () {
    statusFld.textContent = new Date().toISOString() + ": Connection closed";
    promptFld.textContent = "Connection closed";
};
```


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

Sending a Message from a JavaScript WebSocket Client

To send a message from a JavaScript WebSocket client:

1. Define a function for sending the message.
2. In the body of the function for sending the message, call the `send` function of the `WebSocket` object.
3. Call the function that you defined for sending the message.

The following examples show how to send a message from a JavaScript WebSocket client:

- Example 20-37
- Example 20-38

Example 20-37 Defining a Function for Sending a Message

This example defines the function `send` for sending a message.

The creation of the `ws` `WebSocket` object in this example is shown in Example 20-32.

```
function send(message) {
    if (ws && ws.readyState === 1) {
        ws.send(message);
    } else {
        statusFld.textContent = "Connection is not open";
    }
}
```

readyState 1 means open for both the native and fallback APIs. The guard handles the local state check; a connection can still fail after it, so production code must also handle send errors and apply its queue/backpressure policy.

Example 20-38 Calling a Function for Sending a Message

This example calls the `send` function for sending the contents of the text field when the user clicks Send.

The definition of the `send` function is shown in Example 20-37.

```
...
<input type="text" id="input" style="border-radius:2px; border:1px solid #ccc;
margin-top:10px; padding:5px; width:400px;"
placeholder="Type your message here"/>
<button onclick="javascript:send(document.getElementById('input').value);">Send</button>
...
```

Packaging and Specifying the Location of the WebSocket Fallback Client Library

Package the `orasocket.min.js` file in the `scripts` directory of your web application.

In the client application, add the following `script` element to specify the location of `orasocket.min.js`.

```
<script type="text/javascript" src="scripts/orasocket.min.js"></script>
```


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

Enabling WebSocket Fallback

By default, WebSocket fallback is disabled.

To enable WebSocket fallback, set the `com.oracle.tyrus.fallback.enabled` context parameter to `true` in the application's deployment descriptor file `web.xml`.

```
<?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">
  <context-param>
    <param-name>com.oracle.tyrus.fallback.enabled</param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

These are separate minimal web.xml examples. If using both features, merge their context-param elements into one existing web-app root, retaining the application’s other configuration; do not paste two roots or discard existing security settings.

Migrating an Application to the JSR 356 Java API for WebSocket from the Deprecated API

To ensure compatibility of your WebSocket applications with future releases of WebLogic Server, use the JSR 356 Java API for WebSocket instead of the deprecated packages.

As of WebLogic Server 12.1.3, the packages `weblogic.websocket` and `weblogic.websocket.annotation` are deprecated and will be removed in a future release. After these packages have been removed, you will no longer be able to use these packages for connections over the WebSocket protocol.

Comparison of the Jakarta WebSocket API and Proprietary WebLogic Server WebSocket API

Table 20-9 shows the proprietary WebLogic Server WebSocket API and the corresponding Jakarta WebSocket API to use to perform tasks for developing a WebSocket application. The table shows only the Jakarta WebSocket API to use for an annotated endpoint. For each task, the table also provides a cross-reference to instructions for performing the task by using the Jakarta WebSocket API.

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

Table 20-9 Comparison of the Jakarta WebSocket API and Proprietary WebLogic Server WebSocket API

Task	Proprietary WebLogic Server WebSocket API	Jakarta WebSocket API	Instructions
Create a server endpoint class	1. `WebSocketListener` interface or `WebSocketAdapter` superclass 2. `WebSocket` annotation	`ServerEndpoint` annotation	Creating an Annotated Endpoint
Handle a connection opened event	`onOpen` method of a `WebSocketListener` object	`OnOpen` annotation on the method that handles the event	Handling a Connection Opened Event
Handle a message received event	One of the following variants of the overloaded `onMessage` method of a `WebSocketListener` object: • For a message that consists of a text data frame: `onMessage(WebSocketConnection connection, String payload)` • For a message that consists of a binary data frame: `onMessage(WebSocketConnection connection, byte[] payload)`	`OnMessage` annotation on the method that handles the event	Handling a Message Received Event
Handle an error event	`onError` method of a `WebSocketListener` object	`OnError` annotation on the method that handles the event	Handling an Error Event
Handle a connection closed event	`onClose` method of a `WebSocketListener` object	`OnClose` annotation on the method that handles the event	Handling a Connection Closed Event

Send a message	One of the following methods of a `WebSocketConnection` object:
	• `send(String message)`
	• `send(byte[] message)`
	• `sendPing`
	• `sendPong`
	• `stream(boolean last, String fragment)`
	• `stream(boolean last, byte[] fragment, int off, int length)`
		1. `Session` interface
		2. One of the following methods of the `Session` object
		`getBasicRemote()`
		`getAsyncRemote()`
		3. One of the following methods of the `RemoteEndpoint.Basic` object or `RemoteEndpoint.Async` object:
		`sendText`
		`sendBinary`
		`sendPing`
		`sendPong`
			Sending a Message to a Single Peer of an Endpoint

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

Table 20-9 (Cont.) Comparison of the Jakarta WebSocket API and Proprietary WebLogic Server WebSocket API

Task	Proprietary WebLogic Server WebSocket API	Jakarta WebSocket API	Instructions
Send a message to all peers connected to an endpoint	1. `getWebSocketContext` method of a `WebSocketConnection` object 2. `getWebSocketConnections` method of the `WebSocketContext` object obtained by the previous call	`getOpenSessions` method of the `Session` object	Sending a Message to All Peers of an Endpoint
Set the maximum message size for a WebSocket connection	`maxMessageSize` element of the `WebSocket` annotation	`maxMessageSize` element of the OnMessage annotation	
Set the idle timeout value for a WebSocket connection	`timeout` element of the `WebSocket` annotation	One of the following APIs: • For an individual connection: `setMaxIdleTimeout` method of the `Session` object • For the entire container: `setDefaultMaxSessionIdleTimeout` method of a `WebSocketContainer` object	
Set the maximum number of open connections on a WebSocket connection	`maxConnections` element of the `WebSocket` annotation	Not supported by Jakarta WebSocket	

Converting a Proprietary WebSocket Server Endpoint to Use the Jakarta WebSocket API

To convert a proprietary WebSocket server endpoint to use the Jakarta WebSocket API:

1. Convert your `WebSocket` class to an annotated server endpoint class.

   Converting a `WebSocket` class to an annotated endpoint class requires fewer changes than converting the `WebSocket` class to a programmatic endpoint class.

   a. Convert the `WebSocket` class to a POJO class by removing the `extends WebSocketAdapter` clause or `implements WebSocketListener` clause from the class declaration.

   b. Replace the `weblogic.websocket.annotation.WebSocket` annotation on the class declaration with the `jakarta.websocket.server.ServerEndpoint` annotation.

      For more information, see Creating an Annotated Endpoint.

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

> **Note**
>
> If the `pathPatterns` element of your existing endpoint contains the `/*` suffix, you must rewrite your code to achieve the same result as the `/*` suffix. For more information, see Replacing the /* Suffix in a Path Pattern String.

2. Annotate the declaration of each method for handling a life cycle event with the annotation that designates the event that the method handles.

   For more information, see Handling Life Cycle Events in an Annotated WebSocket Endpoint.

3. Replace each reference to the `weblogic.websocket.WebSocketConnection` interface with a reference to the `jakarta.websocket.Session` interface.

4. Replace each method invocation on the `WebSocketConnection` object with an invocation of the corresponding method on the `Session` object.

   The proprietary close(int code) and close(int code, String reason) methods take numeric close codes; ClosingMessage supplies constants for those codes. Jakarta Session.close(CloseReason) takes a CloseReason containing a CloseCode and reason.

5. Change each method invocation on a `Session` object to send a message as follows:

   a. Add an invocation of the `getBasicRemote` method or `getAsyncRemote` method to obtain a reference to the object that represents the peer of this endpoint.

   b. Replace the method in the deprecated API for sending the message with the corresponding method in the Jakarta WebSocket API.

      The method of the Jakarta WebSocket API is a method of the `jakarta.websocket.RemoteEndpoint.Basic` object or `jakarta.websocket.RemoteEndpoint.Async` object to which you obtained a reference in the previous step.

      For more information, see Sending a Message.

Deprecated API Method	RemoteEndpoint.Basic Method	RemoteEndpoint.Async Method
send(String message)	sendText(String text)	One of the following methods: sendText(String text) sendText(String text, SendHandler handler)
send(byte[] message)	sendBinary(ByteBuffer data) session.getBasicRemote().sendBinary(java.nio.ByteBuffer.wrap(data));	One of the following methods: sendBinary (ByteBuffer data) sendBinary(ByteBuffer data, SendHandler handler)

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

Deprecated API Method	RemoteEndpoint.Basic Method	RemoteEndpoint.Async Method
sendPing(byte[] message)	sendPing(ByteBuffer applicationData) session.getBasicRemote().sendPing(java.nio.ByteBuffer.wrap(data));	sendPing(ByteBuffer applicationData)
sendPong(byte[] message)	sendPong(ByteBuffer applicationData) session.getBasicRemote().sendPong(java.nio.ByteBuffer.wrap(data));	sendPong(ByteBuffer applicationData)
stream(boolean last, String fragment)	sendText(String partialMessage, boolean isLast) session.getBasicRemote().sendText(text, last);	No corresponding method.
stream(boolean last, byte[] fragment, int off, int length)	sendBinary(ByteBuffer partialByte, boolean isLast) session.getBasicRemote().sendBinary(java.nio.ByteBuffer.wrap(bytes, off, len), last);	No corresponding method.

data/bytes are the original byte arrays, off/len identify the original slice, and last is the original final-fragment flag. Preserve that slice and flag; do not swap the Jakarta argument order. Partial-message send methods are on Basic; Async has no corresponding partial-message overload.

6. Replace references in `import` clauses to classes in the deprecated API with references to the classes in the Jakarta WebSocket API that your endpoint uses.
7. Recompile and re-deploy the application that uses the server endpoint.

Replacing the /* Suffix in a Path Pattern String

The `pathPatterns` element of the `WebSocket` annotation in the deprecated API accepts the `/*` suffix in a path pattern string. The `/*` suffix matches the path pattern with any resource path that starts with the path pattern before the `/*` suffix. For example, the resource path `/ws/chat` is matched by path pattern `/ws/*`.

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

No equivalent to the `/*` suffix exists in the JSR 356 API. If your existing endpoint relies on the `/*` suffix, you must rewrite your code to achieve the same result as the `/*` suffix. How to rewrite your code depends on whether the `/*` suffix represents variable path parameters in an endpoint URI or additional data for an endpoint.

Replacing a /* Suffix that Represents Variable Path Parameters in an Endpoint URI

The `/*` suffix in a path pattern string might represent one or more variable path parameters in an endpoint URI. In this situation, use a URI template instead of the `/*` suffix.

The JSR 356 API supports only level 1 URI templates in which path parameters are clearly separated by slashes (/). Therefore, in the URI template, you must define one variable for expansion for each variable path parameter that replaces the `/*` suffix in your existing endpoint.

For example, if one variable path parameter replaces the `/*` suffix in your existing endpoint, define a URI template similar to the following example:

```
/ws/{param1}
```

The URI `/ws/test` matches the template in the preceding example. The `param1` variable is expanded to `test`.

Similarly, if two variable path parameters replace the `/*` suffix in your existing endpoint, define a URI template similar to the following example:

```
/ws/{param1}/{param2}
```

The URI `/ws/test/chat` matches the template in the preceding example. The `param1` variable is expanded to `test` and the `param2` variable is expanded to `chat`.

For more information, see Specifying a Part of an Endpoint Deployment URI as an Application Parameter.

Replacing a /* Suffix that Represents Additional Data for an Endpoint

The `/*` suffix in a path pattern string might represent additional data for an endpoint that is transferred as part of the URI. In this situation, use query parameters instead of the `/*` suffix.

Use query parameters for appropriate non-sensitive URI data. Encode names and values correctly, honor applicable URI-length limits, and validate parameters before using them. Do not place credentials, session tokens or sensitive application data in URLs that can be logged or retained by intermediaries.

To obtain an endpoint's query parameters, invoke the method of the endpoint's `Session` object that obtains the parameters in the required format:

- To obtain the parameters as a single string that contains the entire query, invoke the `getQueryString` method. See Example 20-39.
- To obtain the parameters as a map that contains a list of query parameters, invoke the `getRequestParameterMap` method. See Example 20-40.

Example 20-39 Obtaining Query Parameters as a Single String

This example obtains the query parameters in the request URI `/echo?foo=bar,baz,mane,padme,hum` as the application output `"# foo=bar,baz,mane,padme,hum"`.

```
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
```


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

```
import jakarta.websocket.server.ServerEndpoint;
...
@ServerEndpoint("/echo")
public class EchoEndpoint {

    @OnOpen
    public void onOpen(Session session) {
        System.out.println("# " + session.getQueryString());
    }

    // ...
}
```

Example 20-40 Obtaining Query Parameters as a Map

This example obtains the query parameters in the request URI `/echo?foo=bar&foo=baz&foo=mane&foo=padme&foo=hum` as the `List<String>` `# [bar, baz, mane, padme, hum]`.

```
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.HandshakeRequest;
import jakarta.websocket.server.ServerEndpoint;
import java.util.List;
import java.util.Map;
...
@ServerEndpoint("/echo")
public class EchoEndpoint {

    @OnOpen
    public void onOpen(Session session) {
        System.out.println("# " + session.getRequestParameterMap().get("foo"));
    }

    // ...
}
```

Example of Converting a Proprietary WebSocket Server Endpoint to Use the Jakarta WebSocket API

Example 20-41 shows how to convert a proprietary WebSocket server endpoint to use the Jakarta WebSocket API from the deprecated API.

The @OnOpen method sets max idle timeout to 0 to preserve the old disabled-idle-timeout behavior. This is a migration-equivalence demonstration; choose an appropriate finite timeout for the application rather than disabling idle cleanup by default.

Example 20-41 Converting a WebSocket Server Endpoint to Use the Jakarta WebSocket API

This example shows the changes that are required to convert a WebSocket server endpoint to use the Jakarta WebSocket API instead of the deprecated API.

The complete current API listing below preserves the old endpoint's /ws response behavior. The old endpoint used timeout=-1; its disabled idle timeout is mapped explicitly in @OnOpen.

```
package examples.websocket;
import java.io.IOException;
import jakarta.websocket.CloseReason;
import jakarta.websocket.OnMessage;
import jakarta.websocket.OnOpen;
```


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

```
import jakarta.websocket.Session;
import jakarta.websocket.server.ServerEndpoint;

// Replaces the proprietary @WebSocket(timeout=-1, pathPatterns={"/ws"}).
@ServerEndpoint("/ws")
public class MessageListener {
    @OnOpen
    public void open(Session session) {
        session.setMaxIdleTimeout(0); // Preserve the old disabled-idle-timeout setting.
    }
    @OnMessage
    public void onMessage(Session session, String payload) throws IOException {
        try {
            session.getBasicRemote().sendText("Message \"" + payload + "\" has been
received by server.");
        } catch (IOException failure) {
            session.close(new CloseReason(CloseReason.CloseCodes.GOING_AWAY, "Going
away."));
        }
    }
}
```

Example of Using the Java API for WebSocket with WebLogic Server

Examine an example in which a server endpoint echoes text that a user has sent from a client. When the user sends a text message, the server appends the text (from server) to the message and sends the message back to the user.

Example 20-42 Using the Java API for WebSocket with WebLogic Server

```
package com.example.websocket.sample.echo;

import java.io.IOException;

import jakarta.websocket.OnError;
import jakarta.websocket.OnMessage;
import jakarta.websocket.OnOpen;
import jakarta.websocket.Session;
import jakarta.websocket.server.ServerEndpoint;

@ServerEndpoint("/echo")
public class EchoEndpoint {

  @OnOpen
  public void onOpen(Session session) throws IOException {
    session.getBasicRemote().sendText("onOpen is invoked.");
  }

  @OnMessage
  public String echo(String message) {
    return message + " (from server)";
  }

  @OnError
  public void onError(Throwable t) {
    t.printStackTrace();
  }

}
```


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

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

A

Enterprise Application Deployment Descriptor Elements

Enterprise applications can use the standard Jakarta EE application descriptor and optional WebLogic-specific extensions.
An enterprise application can use META-INF/application.xml, the standard Jakarta EE application deployment descriptor, and META-INF/weblogic-application.xml, the WebLogic Server-specific application descriptor. An EAR can omit application.xml when the default module-discovery and naming rules provide the required assembly information. Annotation-based component configuration does not replace every application-level descriptor setting. Include application.xml when explicit assembly configuration is needed; see Enterprise Applications.Enterprise Applications

The `weblogic-application.xml` file is also optional if you are not using any WebLogic Server extensions.

weblogic-application.xml Deployment Descriptor Elements

The `weblogic-application.xml` file is the WebLogic Server-specific deployment descriptor extension for the `application.xml` Jakarta EE deployment descriptor. This is where you configure features such as shared Jakarta EE libraries referenced in the application and EJB caching.

The following sections describe many of the individual elements that are defined in the weblogic-application.xml Schema.

The file is located in the `META-INF` subdirectory of the application archive. The following sections describe elements that can appear in the file.

weblogic-application

The `weblogic-application` element is the root element of the application deployment descriptor.

The following table describes the elements you can define within a `weblogic-application` element.

Required and maximum-occurrence values apply within the immediate containing element. An optional parent can be omitted; required children must be present when that parent is used. Follow schema sequence order and any mutually exclusive choices. Unbounded means the schema does not impose a fixed occurrence maximum.

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

Table A-1 weblogic-application Elements

Element	Required?	Maximum per Parent	Description
`<ejb>`	Optional	1	Contains information that is specific to the EJB modules that are part of a WebLogic application. Currently, one can use the `ejb` element to specify one or more application level caches that can be used by the application's entity beans. For more information on the elements you can define within the `ejb` element, see ejb.
`<xml>`	Optional	1	Contains information about parsers and entity mappings for XML processing that is specific to this application. For more information on the elements you can define within the `xml` element, see xml.
`<jdbc-connection-pool>`	Optional	Unbounded	Zero or more. Specifies an application-scoped JDBC connection pool. For more information on the elements you can define within the `jdbc-connection-pool` element, see jdbc-connection-pool.
`<security>`	Optional	1	Specifies security information for the application. For more information on the elements you can define within the `security` element, see security.

`<application-param>`	Optional	Unbounded	Zero or more. Used to specify un-typed parameters that affect the behavior of container instances related to the application. The parameters listed here are currently supported. Also, these parameters in `weblogic-application.xml` can determine the default encoding to be used for requests and for responses.
			• `webapp.encoding.default`—Can be set to a string representing an encoding supported by the JDK. If set, this defines the default encoding used to process servlet requests and servlet responses. This setting is ignored if `webapp.encoding.usevmdefault` is set to `true`. This value is also overridden for request streams by the `input-charset` element of `weblogic.xml`.
			• `webapp.encoding.usevmdefault`—Can be set to `true` or `false`. If `true`, the system property `file.encoding` is used to define the default encoding.
			The following parameter is used to affect the behavior of Web applications that are contained in this application.
			• `webapp.getrealpath.accept_context_path`—This is a compatibility switch that may be set to `true` or `false`. If set to `true`, the context path of Web applications is allowed in calls to the servlet API `getRealPath`.
			Example:
			`<application-param>`
			`<param-name>webapp.encoding.default`
			`</param-name>`
			`<param-value>UTF8</param-value>`
			`</application-param>`
			For more information on the elements you can define within the `application-param` element, see application-param.

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

Table A-1 (Cont.) weblogic-application Elements

Element	Required?	Maximum per Parent	Description

`<classloader-structure>`	Optional	Unbounded	A classloader-structure element allows you to define the organization of classloaders for this application. The declaration represents a tree structure that represents the classloader hierarchy and associates specific modules with particular nodes. A module's classes are loaded by the classloader that its associated with this element.
			Example:
			
			`<classloader-structure>`
			`<module-ref>`
			`<module-uri>ejb1.jar</module-uri>`
			`</module-ref>`
			`</classloader-structure>`
			`<classloader-structure>`
			`<module-ref>`
			`<module-uri>ejb2.jar</module-uri>`
			`</module-ref>`
			`</classloader-structure>`
			
			For more information on the elements you can define within the `classloader-structure` element, see classloader-structure.

`<listener>`	Optional	Unbounded	Zero or more. Used to register user-defined application lifecycle listeners. These are classes that extend the abstract base class `weblogic.application.ApplicationLifecycleListener`. For more information on the elements you can define within the `listener` element, see listener.
`<singleton-service>`	Optional	Unbounded	Zero or more. Used to register user-defined singleton services. These are classes that implement the interface `weblogic.cluster.singleton.SingletonService`. For more information on the elements you can define within the `singleton-service` element, see singleton-service.
`<startup>`	Optional	Unbounded	Zero or more. Used to register user-defined startup classes. For more information on the elements you can define within the `startup` element, see startup. **Note:** Application-scoped startup and shutdown classes have been deprecated as of release 9.0 of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Programming Application Life Cycle Events
`<shutdown>`	Optional	Unbounded	Zero or more. Used to register user defined shutdown classes. For more information on the elements you can define within the `shutdown` element, see shutdown. **Note:** Application-scoped startup and shutdown classes have been deprecated as of release 9.0 of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Programming Application Life Cycle Events.

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

Table A-1 (Cont.) weblogic-application Elements

Element	Required?	Maximum per Parent	Description

`<module>`	Optional	Unbounded	Represents a single WebLogic application module, such as a JMS or JDBC module.
			This element has the following child elements:
			• `name`—The name of the module.
			• `type`—The type of module. Valid values are JMS, JDBC, Interception, or GAR.
			• `path`—The path of the XML file that fully describes the module, relative to the root of the enterprise application.
			The following example shows how to specify a JMS module called `Workflows`, fully described by the XML file `jms/Workflows-jms.xml`:
			`<module>`
			`  <name>Workflows</name>`
			`  <type>JMS</type>`
			`  <path>jms/Workflows-jms.xml</path>`
			`</module>`

`<library-ref>`	Optional	Unbounded	A reference to a shared Java EE library. For more information on the elements you can define within the `library` element, see library-ref.
`<fair-share-request>`	Optional	Unbounded	Specifies a fair share request class, which is a type of Work Manager request class. In particular, a fair share request class specifies the average percentage of thread-use time required to process requests. The `<fair-share-request>` element can take the following child elements: • `name`—The name of the fair share request class. • `fair-share`—An integer representing the average percentage of thread-use time. See Using Work Managers to Optimize Scheduled Work.
`<response-time-request>`	Optional	Unbounded	Specifies a response time request class, which is a type of Work manager class. In particular, a response time request class specifies a response time goal in milliseconds. The `<response-time-request>` element can take the following child elements: • `name`—The name of the response time request class. • `goal-ms`—The integer response time goal. See Using Work Managers to Optimize Scheduled Work.

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

Table A-1 (Cont.) weblogic-application Elements

Element	Required?	Maximum per Parent	Description
`<context-request>`	Optional	Unbounded	Specifies a context request class, which is a type of Work manager class. In particular, a context request class assigns request classes to requests based on context information, such as the current user or the current user's group. The `<context-request>` element can take the following child elements: • `name`—The name of the context request class. • `context-case`—An element that describes the context. The `<context-case>` element can itself take the following child elements: • `user-name` or `group-name`—The user or group to which the context applies. • `request-class-name`—The name of the request class. See Using Work Managers to Optimize Scheduled Work.
`<max-threads-constraint>`	Optional	Unbounded	Specifies a `max-threads-constraint` Work Manager constraint. A Work Manager constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests. The max-threads constraint limits the number of concurrent threads executing requests from the constrained work set. The `<max-threads-constraint>` element can take the following child elements: • `name`—The name of the max-thread-constraint. • Either `count` or `pool-name`—The integer maximum number of concurrent threads, or the name of a connection pool which determines the maximum. See Using Work Managers to Optimize Scheduled Work.
`<min-threads-constraint>`	Optional	Unbounded	Specifies a `min-threads-constraint` Work Manager constraint. A Work Manager constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests. The min-threads constraint guarantees a number of threads the server will allocate to affected requests to avoid deadlocks. The `<min-threads-constraint>` element can take the following child elements: • `name`—The name of the min-thread-constraint. • `count`—The integer minimum number of threads. See Using Work Managers to Optimize Scheduled Work.

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

Table A-1 (Cont.) weblogic-application Elements

Element	Required?	Maximum per Parent	Description
`<capacity>`	Optional	Unbounded	Specifies a `capacity` Work Manager constraint. A Work Manager constraint defines minimum and maximum numbers of threads allocated to execute requests and the total number of requests that can be queued or executing before WebLogic Server begins rejecting requests. The capacity constraint causes the server to reject requests only when it has reached its capacity. The `<capacity>` element can take the following child elements: • `name`—The name of the capacity constraint. • `count`—The integer thread capacity. See Using Work Managers to Optimize Scheduled Work.
`<work-manager>`	Optional	Unbounded	Specifies the Work Manager that is associated with the application. For more information on the elements you can define within the `work-manager` element, see work-manager. See Using Work Managers to Optimize Scheduled Work for detailed information on Work Managers.
`<application-admin-mode-trigger>`	Optional	Unbounded	Specifies the number of stuck threads needed to bring the application into administration mode. You can specify the following child elements: • `max-stuck-thread-time`—The maximum amount of time, in seconds, that a thread should remain stuck. • `stuck-thread-count`—Number of stuck threads that triggers the stuck thread work manager.
`<session-descriptor>`	Optional	Unbounded	Specifies a list of configuration parameters for servlet sessions. For more information on the elements you can define within the `<session-descriptor>` element, see session-descriptor.
`<library-context-root-override>`	Optional	Unbounded	Zero or more. Used to override the context-root of a Web module specified in the deployment descriptor of a library referenced by this application. For more information on the elements you can define within the `<library-context-root-override>` element, see library-context-root-override.
`<component-factory-class-name>`	Optional	1	Used to enable the Spring extension by setting this element to `org.springframework.jee.interfaces.SpringComponentFactory`. This element exists in EJB, Web, and application descriptors. A module-level descriptor overwrites an application-level descriptor. If set to null (default), the Spring extension is disabled.
`<prefer-application-packages>`	Optional	1	Used for filtering ClassLoader configuration. Specifies a list of packages for classes that must always be loaded from the application.

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

Table A-1 (Cont.) weblogic-application Elements

Element	Required?	Maximum per Parent	Description
`<prefer-application-resources>`	Optional	1	Used for filtering ClassLoader configuration. Specifies a list of resources that must always be loaded from the application, even if the resources are found in the system classloader. Note that the resource loading behavior is different from the resource loading behavior when `<prefer-application-packages>` is used. In that case, application resources get a preference over system resources. The resources captured in this element are never looked up in the system classloader.
`<fast-swap>`	Optional	1	Specifies whether FastSwap deployment is used to minimize redeployment since Java classes are redefined in-place without reloading the ClassLoader. See Using FastSwap Deployment to Minimize Redeployment in *Deploying Applications to Oracle WebLogic Server*. For information on the elements you can define within the `<fast-swap>` element, see fast-swap.
`<ready-registration>`	Optional	1	To use the ReadyApp framework, register an EAR-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 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. For more information, see *Deploying Applications to Oracle WebLogic Server*.

ejb

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

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

Table A-2 ejb Elements

Element	Required?	Maximum per parent	Description

`<entity-cache>`	Optional	Unbounded	Zero or more. The `entity-cache` element is used to define a named application level cache that is used to cache entity EJB instances at runtime. Individual entity beans refer to the application-level cache that they must use, referring to the cache name. There is no restriction on the number of different entity beans that may reference an individual cache.
			To use application-level caching, you must specify the cache using the `<entity-cache-ref>` element of the `weblogic-ejb-jar.xml` descriptor. Two default caches named `ExclusiveCache` and `MultiVersionCache` are used for this purpose. An application may explicitly define these default caches to specify non-default values for their settings. Note that the caching-strategy cannot be changed for the default caches. By default, a cache uses `max-beans-in-cache` with a value of 1000 to specify its maximum size.
			Example:
			`<entity-cache>`
			`<entity-cache-name>ExclusiveCache</entity-cache-name>`
			`<max-cache-size>`
			`<megabytes>50</megabytes>`
			`</max-cache-size>`
			`</entity-cache>`
			For more information on the elements you can define within the `entity-cache` element, see entity-cache.

`<start-mdbs-with-application>`	Optional	1	Allows you to configure the EJB container to start message-driven beans (MDBs) with the application. If set to true, the container starts MDBs as part of the application. If set to false, the container keeps MDBs in a queue and the server starts them as soon as it has started listening on the ports.

entity-cache

The following table describes the elements you can define within an entity-cache element.

Table A-3 entity-cache Elements

Element	Required?	Maximum per parent	Description
`<entity-cache-name>`	Required	1	Specifies a unique name for an entity bean cache. The name must be unique within an ear file and may not be the empty string. Example: `<entity-cache-name>ExclusiveCache</entity-cache-name>`

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

Table A-3 (Cont.) entity-cache Elements

Element	Required?	Maximum per parent	Description
`<max-beans-in-cache>`	Optional If you specify this element, you cannot also specify `<max-cache-size>`.	1	Specifies the maximum number of entity beans that are allowed in the cache. If the limit is reached, beans may be passivated. This mechanism does not take into account the actual amount of memory that different entity beans require. This element can be set to a value of 1 or greater. Default Value: `1000`
`<max-cache-size>`	Optional If you specify this element, you cannot also specify `<max-beans-in-cache>`.	1	Used to specify a limit on the size of an entity cache in terms of memory size—expressed either in terms of bytes or megabytes. A bean provider should provide an estimate of the average size of a bean in the `weblogic-ejb-jar.xml` descriptor if the bean uses a cache that specifies its maximum size using the `max-cache-size` element. By default, a bean is assumed to have an average size of 100 bytes. For more information on the elements you can define within the max-cache-size element, see max-cache-size.
`<max-queries-in-cache>`	Optional	1	Maximum number of query-result cache entries in the application-level query cache. Default: 100.
`<caching-strategy>`	Optional	1	Specifies the general strategy that the EJB container uses to manage entity bean instances in a particular application level cache. A cache buffers entity bean instances in memory and associates them with their primary key value. The `caching-strategy` element can only have one of the following values: • `Exclusive`—Caches a single bean instance in memory for each primary key value. This unique instance is typically locked using the EJB container's exclusive locking when it is in use, so that only one transaction can use the instance at a time. • `MultiVersion`—Caches multiple bean instances in memory for a given primary key value. Each instance can be used by a different transaction concurrently. Default Value: `MultiVersion` Example: `<caching-strategy>Exclusive</caching-strategy>`

max-cache-size

The following table describes the elements you can define within a `max-cache-size` element.

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

Table A-4 max-cache-size Elements

Element	Required?	Maximum Number in File	Description
`<bytes>`	You *must* specify either `<bytes>` or `<megabytes>`	1	The size of an entity cache in terms of memory size, expressed in bytes.
`<megabytes>`	You *must* specify either `<bytes>` or `<megabytes>`	1	The size of an entity cache in terms of memory size, expressed in megabytes.

xml

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

Table A-5 xml Elements

Element	Required?	Maximum per Parent	Description
`<parser-factory>`	Optional	1	The parent element used to specify a particular XML parser or transformer for an enterprise application. For more information on the elements you can define within the `parser-factory` element, see parser-factory.
`<entity-mapping>`	Optional	Unbounded	Zero or More. Specifies the entity mapping. This mapping determines the alternative entity URI for a given public or system ID. The default place to look for this entity URI is the `lib/xml/registry` directory. For more information on the elements you can define within the `entity-mapping` element, see entity-mapping.

parser-factory

The following table describes the elements you can define within a `parser-factory` element.

Table A-6 parser-factory Elements

Element	Required?	Maximum Number in File	Description
`<saxparser-factory>`	Optional	1	Allows you to set the SAXParser Factory for the XML parsing required in this application only. This element determines the factory to be used for SAX style parsing. If you do not specify the `saxparser-factory` element setting, the configured SAXParser Factory style in the Server XML Registry is used. Default Value: Server XML Registry setting
`<document-builder-factory>`	Optional	1	Allows you to set the Document Builder Factory for the XML parsing required in this application only. This element determines the factory to be used for DOM style parsing. If you do not specify the `document-builder-factory` element setting, the configured DOM style in the Server XML Registry is used. Default Value: Server XML Registry setting

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

Table A-6 (Cont.) parser-factory Elements

Element	Required?	Maximum Number in File	Description
`<transformer-factory>`	Optional	1	Allows you to set the Transformer Engine for the style sheet processing required in this application only. If you do not specify a value for this element, the value configured in the Server XML Registry is used. Default value: Server XML Registry setting.

entity-mapping

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

Table A-7 entity-mapping Elements

Element	Required ?	Maximum per parent	Description
`<entity-mapping-name>`	Required	1	Specifies the name for this entity mapping.
`<public-id>`	Optional	1	Specifies the public ID of the mapped entity.
`<system-id>`	Optional	1	Specifies the system ID of the mapped entity.
`<entity-uri>`	Optional	1	Specifies the entity URI for the mapped entity.
`<when-to-cache>`	Optional	1	Legal values are: • cache-on-reference • cache-at-initialization • cache-never The default value is `cache-on-reference`.
`<cache-timeout-interval>`	Optional	1	Specifies the integer value in seconds.

jdbc-connection-pool

> **Note**
>
> The `jdbc-connection-pool` element is deprecated. To define a data source in your enterprise application, you can package a JDBC module with the application. See Configuring JDBC Application Modules for Deployment in *Administering JDBC Data Sources for Oracle WebLogic Server*.

The following table describes the elements you can define within a `jdbc-connection-pool` element.

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

Table A-8 jdbc-connection-pool Elements

Element	Required?	Maximum per parent	Description
`<data-source-jndi-name>`	Required	1	Specifies the JNDI name in the application-specific JNDI tree.
`<connection-factory>`	Required	1	Required connection-factory element. Its optional children, in order, are factory-name and connection-properties. Put user-name, password, url, driver-class-name and connection-params inside connection-properties; see connection-factory below.
`<pool-params>`	Optional	1	Defines parameters that affect the behavior of the pool. For more information on the elements you can define within the `pool-params` element, see pool-params.
`<driver-params>`	Optional	1	Sets behavior on WebLogic Server drivers. For more information on the elements you can define within the `driver-params` element, see driver-params.
`<acl-name>`	Optional	1	DEPRECATED.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-application xmlns="http://xmlns.oracle.com/weblogic/weblogic-
application">
  <jdbc-connection-pool>
    <data-source-jndi-name>jdbc/LegacyPool</data-source-jndi-name>
    <connection-factory>
      <connection-properties>
        <connection-params>
          <parameter>
            <description>Illustrative driver property</description>
            <param-name>foo</param-name>
            <param-value>xyz</param-value>
          </parameter>
        </connection-params>
      </connection-properties>
    </connection-factory>
  </jdbc-connection-pool>
</weblogic-application>
```

Legacy-format nesting illustration. For new definitions follow the retained JDBC Application Modules reference. Replace foo with a property recognized by the actual driver.

connection-factory

The following table describes the elements you can define within a `connection-factory` element.

Table A-9 connection-factory Elements

Element	Required?	Maximum per parent	Description
`<factory-name>`	Optional	1	Specifies the name of a `JDBCDataSourceFactoryMBean` in the `config.xml` file.

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

Table A-9 (Cont.) connection-factory Elements

Element	Required?	Maximum per parent	Description

`<connection-properties>`	Optional	1	Specifies the connection properties for the connection factory. Elements that can be defined for the `connection-properties` element are:
			• `user-name`—Optional. Used to override UserName in the JDBCDataSourceFactoryMBean.
			• `password`—Optional. Used to override Password in the JDBCDataSourceFactoryMBean.
			• `url`—Optional. Used to override URL in the JDBCDataSourceFactoryMBean.
			• `driver-class-name`—Optional. Used to override DriverName in the JDBCDataSourceFactoryMBean
			• `connection-params`—Zero or more. Used to set parameters which will be passed to the driver when making a connection. Example:
			`<connection-params>`
			`<parameter>`
			`<description>Desc of param`
			`</description>`
			`<param-name>foo</param-name>`
			`<param-value>xyz</param-value>`
			`</parameter>`
			`</connection-params>`
			Each connection-params contains zero or more parameter elements. Each parameter contains optional description, then required param-name and required param-value, in that order.

pool-params

The following table describes the elements you can define within a `pool-params` element.

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

Table A-10 pool-params Elements

Element	Required?	Maximum Number in File	Description
<size-params>	Optional	1	Defines parameters that affect the number of connections in the pool. • initial-capacity—Optional. The initial-capacity element defines the number of physical database connections to create when the pool is initialized. The default value is 1. • max-capacity—Optional. The max-capacity element defines the maximum number of physical database connections that this pool can contain. Note that the JDBC Driver may impose further limits on this value. The default value is 1. • capacity-increment—Optional. The capacity-increment element defines the increment by which the pool capacity is expanded. When there are no more available physical connections to service requests, the pool creates this number of additional physical database connections and adds them to the pool. The pool ensures that it does not exceed the maximum number of physical connections as set by max-capacity. The default value is 1. • shrinking-enabled—Optional. The shrinking-enabled element indicates whether or not the pool can shrink back to its initial-capacity when connections are detected to not be in use. • shrink-period-minutes—Optional. The shrink-period-minutes element defines the number of minutes to wait before shrinking a connection pool that has incrementally increased to meet demand. The shrinking-enabled element must be set to true for shrinking to take place. • shrink-frequency-seconds—Optional. • highest-num-waiters—Optional. • highest-num-unavailable—Optional.
<xa-params>	Optional	1	Defines the parameters for the XA DataSources. • debug-level—Optional. Integer. The debug-level element defines the debugging level for XA operations. The default value is 0. • keep-conn-until-tx-complete-enabled—Optional. Boolean. If you set the keep-conn-until-tx-complete-enabled element to true, the XA connection pool associates the same XA connection with the distributed transaction until the transaction completes. • end-only-once-enabled—Optional. Boolean. If you set the end-only-once-enabled element to true, the XAResource.end() method is only called once for each pending XAResource.start() method. • recover-only-once-enabled—Optional. Boolean. If you set the recover-only-once-enabled element to true, recover is only called one time on a resource. • tx-context-on-close-needed—Optional. Set the tx-context-on-close-needed element to true if the XA driver requires a distributed transaction context when closing various JDBC objects (for example, result sets, statements, connections, and so on). If set to true, the SQL exceptions that are thrown while closing the JDBC objects in no transaction context are swallowed. • new-conn-for-commit-enabled—Optional. Boolean. If you set the new-conn-for-commit-enabled element to true, a dedicated XA connection is used for commit/rollback processing of a particular distributed transaction.

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

Table A-10 (Cont.) pool-params Elements

Element	Required?	Maximum Number in File	Description
<xa-params> Continued...	Optional	1	• prepared-statement-cache-size—**Deprecated**. Optional. Use the prepared-statement-cache-size element to set the size of the prepared statement cache. The size of the cache is a number of prepared statements created from a particular connection and stored in the cache for further use. Setting the size of the prepared statement cache to 0 turns it off. **Note:** Prepared-statement-cache-size is deprecated. Use cache-size in driver-params/prepared-statement. See driver-params for more information. • keep-logical-conn-open-on-release—Optional. Boolean. Set the keep-logical-conn-open-on-release element to true, to keep the logical JDBC connection open when the physical XA connection is returned to the XA connection pool. The default value is false. • local-transaction-supported—Optional. Boolean. Set the local-transaction-supported to true if the XA driver supports SQL with no global transaction; otherwise, set it to false. The default value is false. • resource-health-monitoring-enabled—Optional. Set the resource-health-monitoring-enabled element to true to enable JTA resource health monitoring for this connection pool.

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

Table A-10 (Cont.) pool-params Elements

Element	Required?	Maximum Number in File	Description
<xa-params> Continued...	Optional	1	• xa-set-transaction-timeout—Optional. Used in: xa-params Example: <xa-set-transaction-timeout> true </xa-set-transaction-timeout> • xa-transaction-timeout—Optional. When the xa-set-transaction-timeout value is set to true, the transaction manager invokes setTransactionTimeout on the resource before calling XAResource.start. The Transaction Manager passes the global transaction timeout value. If this attribute is set to a value greater than 0, then this value is used in place of the global transaction timeout. Default value: 0 Used in: xa-params Example: <xa-transaction-timeout> 30 </xa-transaction-timeout> • rollback-localtx-upon-connclose—Optional. When the rollback-localtx-upon-connclose element is true, the connection pool calls rollback() on the connection before putting it back in the pool. Default value: false Used in: xa-params Example: <rollback-localtx-upon-connclose> true </rollback-localtx-upon-connclose>
<login-delay-seconds>	Optional	1	Sets the number of seconds to delay before creating each physical database connection. Some database servers cannot handle multiple requests for connections in rapid succession. This property allows you to build in a small delay to let the database server catch up. This delay occurs both during initial pool creation and during the lifetime of the pool whenever a physical database connection is created.
<leak-profiling-enabled>	Optional	1	Enables JDBC connection leak profiling. A connection leak occurs when a connection from the pool is not closed explicitly by calling the close() method on that connection. When connection leak profiling is active, the pool stores the stack trace at the time the connection object is allocated from the pool and given to the client. When a connection leak is detected (when the connection object is garbage collected), this stack trace is reported. This element uses extra resources and will likely slowdown connection pool operations, so it is not recommended for production use.

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

Table A-10 (Cont.) pool-params Elements

Element	Required?	Maximum Number in File	Description
`<connection-check-params>`	Optional	1	• Defines whether, when, and how connections in a pool is checked to make sure they are still alive. • `table-name`—Optional. The `table-name` element defines a table in the schema that can be queried. • `check-on-reserve-enabled`—Optional. If the check-on-reserve-enabled element is set to true, then the connection will be tested each time before it is handed out to a user. • `check-on-release-enabled`—Optional. If the `check-on-release-enabled` element is set to `true`, then the connection will be tested each time a user returns a connection to the pool. • `refresh-minutes`—Optional. If the `refresh-minutes` element is defined, a trigger is fired periodically (based on the number of minutes specified). This trigger checks each connection in the pool to make sure it is still valid. • `check-on-create-enabled`—Optional. If set to `true`, then the connection will be tested when it is created. • `connection-reserve-timeout-seconds`—Optional. Number of seconds after which the call to reserve a connection from the pool will timeout. • `connection-creation-retry-frequency-seconds`—Optional. The frequency of retry attempts by the pool to establish connections to the database. • `inactive-connection-timeout-seconds`—Optional. The number of seconds of inactivity after which reserved connections will forcibly be released back into the pool.
`<connection-check-params>` `Continued...`	Optional	1	• `test-frequency-seconds`—Optional. The number of seconds between database connection tests. After every test-frequency-seconds interval, unused database connections are tested using `table-name`. Connections that do not pass the test will be closed and reopened to re-establish a valid physical database connection. If `table-name` is not set, the test will not be performed. • `init-sql`—Optional. Specifies a SQL query that automatically runs when a connection is created.
`<jdbcxa-debug-level>`	Optional	1	This is an internal setting.
`<remove-infected-connections-enabled>`	Optional	1	Controls whether a connection is removed from the pool when the application asks for the underlying vendor connection object. Enabling this attribute has an impact on performance; it essentially disables the pooling of connections (as connections are removed from the pool and replaced with new connections).

driver-params

The following table describes the elements you can define within a `driver-params` element.

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

Table A-11 driver-params Elements

Element	Required?	Maximum per parent	Description

`<statement>`	Optional	1	Defines the `driver-params` statement. Contains the following optional element: `profiling-enabled`.
			Example:
			`<statement>`
			`<profiling-enabled>true`
			`</profiling-enabled>`
			`</statement>`

`<prepared-statement>`	Optional	1	Enables the running of JDBC prepared statement cache profiling. When enabled, prepared statement cache profiles are stored in external storage for further analysis. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. The default value is false. • `profiling-enabled`—Optional. • `cache-profiling-threshold`—Optional. The `cache-profiling-threshold` element defines a number of statement requests after which the state of the prepared statement cache is logged. This element minimizes the output volume. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. • `cache-size`—Optional. The `cache-size` element returns the size of the prepared statement cache. The size of the cache is a number of prepared statements created from a particular connection and stored in the cache for further use. • `parameter-logging-enabled`—Optional. During SQL roundtrip profiling it is possible to store values of prepared statement parameters. The `parameter-logging-enabled` element enables the storing of statement parameters. This is a resource-consuming feature, so it is recommended that you turn it off on a production server. Parameter logging can record sensitive application data. Enable it only for a controlled diagnostic purpose, protect access to the resulting logs and apply an appropriate retention policy. Limiting max-parameter-length limits output length; it does not redact sensitive values. • `max-parameter-length`—Optional. During SQL roundtrip profiling it is possible to store values of prepared statement parameters. The `max-parameter-length` element defines maximum length of the string passed as a parameter for JDBC SQL roundtrip profiling. This is a resource-consuming feature, so you should limit the length of data for a parameter to reduce the output volume. • `cache-type`—Optional.
`<row-prefetch-enabled>`	Optional	1	Specifies whether to enable row prefetching between a client and WebLogic Server for each ResultSet. When an external client accesses a database using JDBC through Weblogic Server, row prefetching improves performance by fetching multiple rows from the server to the client in one server access. WebLogic Server ignores this setting and does not use row prefetching when the client and WebLogic Server are in the same JVM

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

Table A-11 (Cont.) driver-params Elements

Element	Required?	Maximum per parent	Description
`<row-prefetch-size>`	Optional	1	Specifies the number of result set rows to prefetch for a client. The optimal value depends on the particulars of the query. In general, increasing this number increases performance, until a particular value is reached. At that point further increases do not result in any significant increase in performance. **Note:** Typically you will not see any increase in performance after 100 rows. The default value should be adequate for most situations. Valid values for this element are between 2 and 65536. The default value is 48.
`<stream-chunk-size>`	Optional	1	Specifies the data chunk size for streaming data types, which are pulled from WebLogic Server to the client as needed.

security

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

Table A-12 security Elements

Element	Required?	Maximum per parent	Description
`<realm-name>`	Optional	1	Deprecated and ignored since WebLogic Server 12.2.1.0.0. The applicable realm is determined by the deployment target; this element does not select a different realm for the application.

`<security-role-assignment>`	Optional	Unbounded	Declares a mapping between an application-wide security role and one or more WebLogic Server principals.
			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>`
			Within security-role-assignment, role-name is required. Follow it with either one or more principal-name elements or one externally-defined element, not both. externally-defined indicates that the role mapping is supplied externally to the application descriptor.

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

PayrollAdmin is an illustrative application role. Retain deployment security-mode prerequisites and realm mappings; this descriptor does not create users, groups or realms.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-application xmlns="http://xmlns.oracle.com/weblogic/weblogic-
application">
  <security><security-role-assignment><role-name>PayrollAdmin</role-
name><externally-defined/></security-role-assignment></security>
</weblogic-application>
```

application-param

The following table describes the elements you can define within an application-param element.

Table A-13 application-param Elements

Element	Required?	Maximum per parent	Description
`<description>`	Optional	1	Provides a description of the application parameter.
`<param-name>`	Required	1	Defines the name of the application parameter.
`<param-value>`	Required	1	Defines the value of the application parameter.

classloader-structure

The following table describes the elements you can define within a `classloader-structure` element.

Table A-14 classloader-structure Elements

Element	Required?	Maximum per parent	Description
`<module-ref>`	Optional	Unbounded	The following list describes the elements you can define within a `module-ref` element: • module-uri—Required, exactly one within each module-ref. Identifies the URI of one module in the enterprise application. Use additional module-ref elements for additional modules.
`<classloader-structure>`	Optional	Unbounded	Allows for arbitrary nesting of classloader structures for an application. However, for this version of WebLogic Server, the depth is restricted to three levels.

listener

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

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

Table A-15 listener Elements

Element	Required?	Maximum Number in File	Description
`<listener-class>`	Required	1	Name of the user's implementation of `ApplicationLifecycleListener`.
`<listener-uri>`	Optional	1	A JAR file within the EAR that contains the implementation. If you do not specify the `listener-uri`, it is assumed that the class is visible to the application.
`<run-as-principal-name>`	Optional	1	Specific a user identity to startup and shutdown application lifecycle events. The identity specified here should be a valid user name in the system. If `run-as-principal-name` is not specified, the deployment initiator user identity will be used as the `run-as` identity for the execution of the application lifecycle listener. **Note:** If the `run-as-principal-name` identity defined for the application lifecycle listener is an administrator, the application deployer must have administrator privileges; otherwise, deployment will fail.

singleton-service

The following table describes the elements you can define within a `singleton-service` element.

Table A-16 singleton-service Elements

Element	Required?	Maximum per parent	Description
`<class-name>`	Required	1	Fully qualified implementation class of weblogic.cluster.singleton.SingletonService, whose activate/deactivate callbacks represent service ownership.
<name>	Required	1	Name identifying this singleton service.
`<singleton-uri>`	Optional	1	Defines a JAR file within the EAR that contains the `singleton-service`. If `singleton-uri` is not defined, then it is assumed that the class is visible to the application.

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

Replace the class, service name and JAR path with your application values; package the implementation in the EAR. See the singleton-service description for the clustering/service-lifecycle context.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-application xmlns="http://xmlns.oracle.com/weblogic/weblogic-
application">
  <singleton-service>
    <class-name>com.example.MySingletonService</class-name>
    <name>MySingletonService</name>
    <singleton-uri>services.jar</singleton-uri>
  </singleton-service>
</weblogic-application>
```

startup

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

> **Note**
>
> Application-scoped startup and shutdown classes have been deprecated as of release 9.0 of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Programming Application Life Cycle Events.

Table A-17 startup Elements

Element	Required?	Maximum Number in File	Description
`<startup-class>`	Required	1	Defines the name of the class to be run when the application is being deployed.
`<startup-uri>`	Optional	1	Defines a JAR file within the EAR that contains the `startup-class`. If `startup-uri` is not defined, then its assumed that the class is visible to the application.

shutdown

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

> **Note**
>
> Application-scoped startup and shutdown classes have been deprecated as of release 9.0 of WebLogic Server. Instead, you should use lifecycle listener events in your applications. For details, see Programming Application Life Cycle Events.

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

Table A-18 shutdown Elements

Element	Required?	Maximum per parent	Description
`<shutdown-class>`	Required	1	Defines the name of the class to be run when the application is undeployed.
`<shutdown-uri>`	Optional	1	Defines a JAR file within the EAR that contains the `shutdown-class`. If you do not define the `shutdown-uri` element, it is assumed that the class is visible to the application.

work-manager

The following table describes the elements you can define within a work-manager element.

See Using Work Managers to Optimize Scheduled Work for examples and information on Work Managers.

Table A-19 work-manager Elements

Element	Required ?	Maximum per parent	Description
`<name>`	Required	1	The name of the Work Manager.
`<response-time-request-class>`	Optional	1	See the description of the `<response-time-request>` element in weblogic-application for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<fair-share-request-class>`, `<context-request-class>`, or `<request-class-name>`.
`<fair-share-request-class>`	Optional	1	See the description of the `<fair-share-request>` element in weblogic-application for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<response-time-request-class>`, `<context-request-class>`, or `<request-class-name>`.
`<context-request-class>`	Optional	1	See the description of the `<context-request>` element in weblogic-application for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<fair-share-request-class>`, `<response-time-request-class>`, or `<request-class-name>`.
`<request-class-name>`	Optional	1	The name of the request class. If you specify this element, you cannot also specify `<fair-share-request-class>`, `<context-request-class>`, or `<response-time-request-class>`.
`<min-threads-constraint>`	Optional	1	See the description of the `<min-threads-constraint>` element in weblogic-application for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<min-threads-constraint-name>`.

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

Table A-19 (Cont.) work-manager Elements

Element	Required ?	Maximum per parent	Description
`<min-threads-constraint-name>`	Optional	1	The name of the min-threads constraint. If you specify this element, you cannot also specify `<min-threads-constraint>`.
`<max-threads-constraint>`	Optional	1	See the description of the `<max-threads-constraint>` element in weblogic-application for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<max-threads-constraint-name>`.
`<max-threads-constraint-name>`	Optional	1	The name of the max-threads constraint. If you specify this element, you cannot also specify `<max-threads-constraint>`.
`<capacity>`	Optional	1	See the description of the `<capacity>` element in weblogic-application for information on this child element of `<work-manager>`. If you specify this element, you cannot also specify `<capacity-name>`.
`<capacity-name>`	Optional	1	The name of the capacity constraint for queued or executing requests. If you specify this element, you cannot also specify `<capacity>`.
`<work-manager-shutdown-trigger>`	Optional	1	Used to specify a Stuck Thread Work Manager component that can shut down the Work Manager in response to stuck threads. You can specify the following child elements: • max-stuck-thread-time—Optional execution-duration threshold, in seconds, used to classify a thread as stuck for this action. When no positive override is specified, the server's applicable stuck-thread threshold is used. • stuck-thread-count—Required number of stuck threads that triggers shutdown of this Work Manager. If you specify this element, you cannot also specify `<ignore-stuck-threads>`.
`<ignore-stuck-threads>`	Optional	1	Specifies whether the Work Manager should ignore stuck threads and never shut down even if threads become stuck. If you specify this element, you cannot also specify `<work-manager-shutdown-trigger>`.

session-descriptor

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

This table covers selected session settings. Application- and module-level descriptor settings and standard Servlet session configuration can affect the effective values; see Using Sessions and Session Persistence for precedence and mode prerequisites.

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

Table A-20 session-descriptor Elements

Element	Required?	Maximum per parent	Description
`<timeout-secs>`	Optional	1	Specifies the number of seconds after which the session times out. Default value is 3600 seconds.
`<invalidation-interval-secs>`	Optional	1	Specifies the number of seconds of the invalidation trigger interval. Default value is 60 seconds.
<max-save-post-size>	Optional	1	Maximum saved POST size for FORM authentication, in bytes. Default: 4096.
<save-post-timeout-secs>	Optional	1	Saved-POST session timeout for FORM authentication, in seconds. Default: 40.
<save-post-timeout-interval-secs>	Optional	1	Saved-POST session invalidation-check interval, in seconds. Default: 20.
`<debug-enabled>`	Optional	1	Specifies whether debugging is enabled for HTTP sessions. Default value is `false`.
`<id-length>`	Optional	1	Specifies the length of the session ID. Default value is 52.
<auth-cookie-id-length>	Optional	1	Authentication-cookie identifier length. Default: 20; minimum: 8.
`<tracking-enabled>`	Optional	1	Specifies whether session tracking is enabled between HTTP requests. Default value is `true`.
`<cache-size>`	Optional	1	Specifies the cache size for JDBC and file persistent sessions. Default: 1024.
`<max-in-memory-sessions>`	Optional	1	Specifies the maximum sessions limit for memory/replicated sessions. Default value is -1, or unlimited.
`<cookies-enabled>`	Optional	1	Specifies the Web application container should set cookies in the response. Default value is `true`.
`<cookie-name>`	Optional	1	Specifies the name of the cookie that tracks sessions. Default name is `JSESSIONID`.
`<cookie-path>`	Optional	1	Specifies the session tracking cookie path. Default value is `/`.
`<cookie-domain>`	Optional	1	Specifies the session tracking cookie domain. Default value is `null`.
`<cookie-comment>`	Optional	1	Specifies the session tracking cookie comment. Default value is `null`.
`<cookie-secure>`	Optional	1	Specifies whether the session tracking cookie is marked secure. Default value is `false`.
`<cookie-max-age-secs>`	Optional	1	Maximum age of the session-tracking cookie, in seconds. Default: -1, creating a nonpersistent session cookie. A positive value sets a persistent lifetime; zero requests deletion. Cookie lifetime is separate from the server-side HTTP-session timeout.
<cookie-http-only>	Optional	1	Marks the session-tracking cookie HttpOnly. Default: true.

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

Table A-20 (Cont.) session-descriptor Elements

Element	Required?	Maximum per parent	Description
<persistent-store-type>	Optional	1	Specifies the type of storage for session persistence. You can specify the following values: • memory—Default value. • replicated—Requires clustering. • replicated_if_clustered—Defaults to memory in non-clustered case. • file • jdbc • cookie • async-replicated—Asynchronous session replication; requires clustering • async-replicated-if-clustered—Asynchronous replication when clustered, otherwise memory • async-jdbc—Asynchronous JDBC persistence; requires the configured persistent data-source JNDI name and database setup • coherence-web—Coherence*Web integration; requires its session SPI library and Coherence runtime configuration These modes have different deployment, persistence and failover requirements; configure the chosen mode as described in Using Sessions and Session Persistence. This list does not establish that optional integration libraries are installed. Using Sessions and Session Persistence
<extended-session-format>	Optional	1	Extended session-ID format: none, plugin or host-server-names. Default: none.
<persistent-store-cookie-name>	Optional	1	Specifies the name of the cookie that holds the attribute name and values when using cookie-based session persistence. Default value is WLCOOKIE.
<persistent-store-dir>	Optional	1	Specifies the name of the directory when using file-based session persistence. The directory is relative to the temporary directory defined for the Web application. Default value is session_db.
<persistent-store-pool>	Optional	1	Specifies the name of the JDBC connection pool when using jdbc-based session persistence.
<persistent-data-source-jndi-name>	Optional	1	JNDI name of the data source used for JDBC/async-jdbc session persistence; required operational configuration for async-jdbc.
<persistent-session-flush-interval>	Optional	1	Asynchronous JDBC session flush interval, in seconds. Default: 180.
<persistent-session-flush-threshold>	Optional	1	Asynchronous JDBC session flush threshold. Default: 100.
<persistent-async-queue-timeout>	Optional	1	Asynchronous JDBC persistence queue timeout, in seconds. Default: 30.
<persistent-store-table>	Optional	1	Specifies the name of the database table when using jdbc-based session persistence. Default value is wl_servlet_sessions.

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

Table A-20 (Cont.) session-descriptor Elements

Element	Required?	Maximum per parent	Description
`<jdbc-column-name-max-inactive-interval>`	Optional	1	Alternative name for the `wl_max_inactive_interval` column name when using `jdbc`-based session persistence. Required for certain databases that do not support long column names
`<jdbc-connection-timeout-secs>`	Optional	1	DEPRECATED
`<url-rewriting-enabled>`	Optional	1	Controls URL-based session tracking. Default: false. Explicit application tracking-mode configuration can enable URL tracking. When tracking modes are implicit, the implementation can still propagate a session ID in responses before cookie support has been established; explicitly setting url-rewriting-enabled to false suppresses that bootstrap fallback. This setting does not change cookie persistence or the server-side session timeout.
`<http-proxy-caching-of-cookies>`	Optional	1	Specifies whether WebLogic Server adds the following HTTP header to the response: `Cache-control: no-cache=set-cookie` This header specifies that proxy caches should not cache the cookies. Default value is `true`, which means that the header is NOT added. Set this element to `false` if you want the header added to the response.
`<encode-session-id-in-query-params>`	Optional	1	When URL encoding of a session ID is performed, true places it in the query string; false uses a semicolon-delimited path parameter. Default: false. This option selects the encoding form; it does not independently enable URL-based session tracking.
`<monitoring-attribute-name>`	Optional	1	Used to tag runtime information for different sessions. For example, set this element to `username` if you have a `username` attribute that is guaranteed to be unique.
`<sharing-enabled>`	Optional	1	Enables HTTP-session sharing among web modules in the same enterprise application (EAR), subject to compatible session/cookie configuration. Default: false. This is not sharing between arbitrary separately deployed applications and does not itself enable cluster replication.
<invalidate-on-relogin>	Optional	1	Invalidates the current session when a logged-in user switches to another user. Default: false.

HttpOnly restricts script access to the cookie. It does not imply Secure or prevent every script-related attack. Defaults here are descriptive, not recommendations to weaken application settings.

library-ref

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

See Creating Shared Java EE Libraries and Optional Packages, for additional information and examples.

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

Table A-21 library Elements

Element	Required?	Maximum Number in File	Description
`<library-name>`	Required	1	Specifies the name of the referenced shared Java EE library.
`<specification-version>`	Optional	1	Specifies the minimum specification-version required.
`<implementation-version>`	Optional	1	Specifies the minimum implementation-version required.
`<exact-match>`	Optional	1	Specifies whether there must be an exact match between the specification and implementation version that is specified and that of the referenced library. Default value is `false`.
`<context-root>`	Optional	1	Specifies the context-root of the referenced Web application's shared Java EE library.

library-context-root-override

The following table describes the elements you can define within a `library-context-root-override` element to override `context-root` elements within a referenced EAR library. See library-ref.

See Creating Shared Jakarta EE Libraries and Optional Packages, for additional information and examples.

The enclosing library-context-root-override element is optional and repeatable under weblogic-application. Each occurrence must contain context-root followed by override-value. Without an override entry, the library context root is retained.

Table A-22 library-context-root-override Elements

Element	Required?	Maximum per parent	Description
`<context-root>`	Required	1	Identifies the existing context root of the web module in a referenced EAR library.
`<override-value>`	Required	1	Supplies the replacement context root to use in the referencing application.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-application xmlns="http://xmlns.oracle.com/weblogic/weblogic-
application">
  <library-context-root-override>
    <context-root>/library</context-root>
    <override-value>/application</override-value>
  </library-context-root-override>
</weblogic-application>
```

fast-swap

The following table describes the elements you can define within a `fast-swap` element.

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

For more information about FastSwap Deployment, see Using FastSwap Deployment to Minimize Redeployment in *Deploying Applications to Oracle WebLogic Server*.

FastSwap is a development-time class-redefinition feature. Use the deployment guide for prerequisites, supported changes and limitations; use normal deployment procedures for production changes.

Table A-23 fast-swap Elements

Element	Required?	Maximum per parent	Description
`<enabled>`	Optional	1	Set to `true` to enable FastSwap deployment in your application. Default within a configured fast-swap element: true. Omitting the entire fast-swap element does not enable the feature.
`<refresh-interval>`	Optional	1	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. Units: seconds. Default: 3.
`<redefinition-task-limit>`	Optional	1	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. Default: 10; minimum: 1. This is the number of task records retained, not a limit on concurrent application requests.

weblogic-application.xml Schema

The WebLogic Server 26.1 application descriptor schema is weblogic-application version 2.0: http://xmlns.oracle.com/weblogic/weblogic-application/2.0/weblogic-application.xsd. The namespace remains http://xmlns.oracle.com/weblogic/weblogic-application.weblogic-application.xsd

These references identify current descriptor formats; they do not require converting every supported older descriptor.

application.xml Schema

For the Jakarta EE 11 application.xml grammar, see application_11.xsd at https://jakarta.ee/xml/ns/jakartaee/application_11.xsd. Its namespace is https://jakarta.ee/xml/ns/jakartaee and its descriptor version is 11.application_11.xsd

These references identify current descriptor formats; they do not require converting every supported older descriptor.Jakarta EE schema index

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

B

wldeploy Ant Task Reference

Use the wldeploy Ant task to perform WebLogic Server deployment operations from an Ant build file.

Overview of the wldeploy Ant Task

The `wldeploy` Ant task enables you to perform `weblogic.Deployer` functions using attributes specified in an Ant XML file.

You can use `wldeploy` along with other WebLogic Server Ant tasks to create a single Ant build script that:

- Builds your application from source, using `wlcompile`, `appc`, and the Web services Ant tasks.
- Creates, starts, and configures a new WebLogic Server domain, using the `wlserver` and `wlconfig` Ant tasks.
- Deploys a compiled application to the newly-created domain, using the `wldeploy` Ant task.

See Using Ant Tasks to Configure and Use a WebLogic Server Domain, for more information about `wlserver` and `wlconfig`. See Building Applications in a Split Development Directory, for information about `wlcompile`.

Basic Steps for Using wldeploy

To use the wldeploy Ant task you must perform several required and some optional steps.

1. Set WL_HOME to the wlserver directory inside the WebLogic Oracle home. In a Bourne-compatible shell such as bash, source the environment script in the same shell that will run Ant. On Windows, call the command script from the command prompt that will run Ant.

   Bourne-compatible shell

   ```
   . "$WL_HOME/server/bin/setWLSEnv.sh"
   ```

   Windows command prompt

   ```
   call "%WL_HOME%\server\bin\setWLSEnv.cmd"
   ```

2. Create build.xml from the complete project in Sample build.xml Files for wldeploy. It explicitly loads WL_HOME/server/lib/weblogic.jar and uses the matching installed Ant runtime.Sample build.xml Files for wldeploy
3. If necessary, add task definitions and calls to the `wlserver` and `wlconfig` tasks in the build script to create and start a new WebLogic Server domain. See Using Ant Tasks to Configure and Use a WebLogic Server Domain, for information about `wlserver` and `wlconfig`.

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

4. Add a call to `wldeploy` to deploy your application to one or more WebLogic Server instances or clusters. See Sample build.xml Files for wldeploy and wldeploy Ant Task Attribute Reference.
5. Run ant help, then choose the named deployment target and supply its documented properties.

   ```
   ant help
   ```

The acting user needs deployment permissions. Domain creation/configuration integration is optional; an already running, configured domain is sufficient.

Sample build.xml Files for wldeploy

Examine these sample `build.xml` files which show how to deploy an application on a single WebLogic Server instance, undeploy the application, perform a partial redeploy of the application, undeploy a particular file in the application, and deploy a Jakarta EE library.

First create a WLST storeUserConfig user configuration/key pair for an identity authorized to deploy, following the WLST command reference. Protect both files; do not commit them or put a real password in this build file. Supply their paths as userconfig.file and userkey.file and omit user/password. Run these examples where the source paths are accessible to the Administration Server; remote and upload path semantics are described in the attribute reference. Configure the client JVM's TLS trust for the selected endpoint. The XML does not create a domain or grant permissions.

The five targets deploy, undeploy, redeploy_partial, delete_static and deploy_library perform distinct operations. app.source identifies an existing compiled EAR/archive or exploded application; library.source identifies the existing library archive/directory. Initial deploy uses stage mode. For partial operations use an exploded EAR containing exploded SimpleImpl.war and static SimpleImpl.war/SimpleImpl.jsp; override module.uri/static.uri with actual application-relative paths. The sample paths are illustrative.

failonerror remains true; a failed operation must fail this build. Enable debugging separately only when diagnosing a problem.

For an exploded application deployed in stage mode, action=redeploy with deleteFiles=true and deltaFiles removes only the listed static files from the targeted staging copies. Supply explicit target servers. This is different from undeploying the named application. Do not combine it with upload, and do not use the example as a general production file-management procedure.

Undeploy identifies the existing deployment by name and target. Redeploy can replace the complete application from a source archive/directory or update selected application-relative files/modules with deltaFiles. In the combined example, the redeploy_partial target uses deltaFiles with the already deployed source and therefore omits source. A full-source redeploy is a supported alternative that this target does not demonstrate.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="wldeploy-example" default="help" basedir=".">
    <property environment="env"/>
    <fail unless="env.WL_HOME" message="Set WL_HOME and source/call setWLSEnv
first."/>
    <taskdef name="wldeploy"
classname="weblogic.ant.taskdefs.management.WLDeploy"
             classpath="${env.WL_HOME}/server/lib/weblogic.jar"/>
    <property name="app.name" value="DeployExample"/>
    <property name="library.name" value="myLibrary"/>
```


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

```
    <property name="module.uri" value="SimpleImpl.war"/>
    <property name="static.uri" value="SimpleImpl.war/SimpleImpl.jsp"/>

    <target name="help">
        <echo>Choose deploy, undeploy, redeploy_partial, delete_static, or
deploy_library.</echo>
        <echo>Supply admin.url, deploy.targets, userconfig.file and
userkey.file with -D.</echo>
        <echo>Supply app.source for deploy, or library.source for
deploy_library.</echo>
    </target>
    <target name="check-connection">
        <fail unless="admin.url" message="Supply -Dadmin.url=..."/>
        <fail unless="deploy.targets" message="Supply -Ddeploy.targets=..."/>
        <fail unless="userconfig.file" message="Supply -
Duserconfig.file=..."/>
        <fail unless="userkey.file" message="Supply -Duserkey.file=..."/>
        <available file="${userconfig.file}" type="file"
property="config.exists"/>
        <available file="${userkey.file}" type="file" property="key.exists"/>
        <fail unless="config.exists" message="The user configuration file
does not exist."/>
        <fail unless="key.exists" message="The user key file does not
exist."/>
    </target>
    <target name="deploy" depends="check-connection">
        <fail unless="app.source" message="Supply -Dapp.source=..."/>
        <wldeploy action="deploy" name="${app.name}" source="${app.source}"
stage="true"
                adminurl="${admin.url}" targets="${deploy.targets}"
                userconfigfile="${userconfig.file}" userkeyfile="$
{userkey.file}"
                failonerror="true" verbose="true"/>
    </target>
    <target name="undeploy" depends="check-connection">
        <wldeploy action="undeploy" name="${app.name}"
                adminurl="${admin.url}" targets="${deploy.targets}"
                userconfigfile="${userconfig.file}" userkeyfile="$
{userkey.file}"
                failonerror="true" verbose="true"/>
    </target>
    <target name="redeploy_partial" depends="check-connection">
        <wldeploy action="redeploy" name="${app.name}" deltaFiles="$
{module.uri}"
                adminurl="${admin.url}" targets="${deploy.targets}"
                userconfigfile="${userconfig.file}" userkeyfile="$
{userkey.file}"
                failonerror="true" verbose="true"/>
    </target>
    <target name="delete_static" depends="check-connection">
        <wldeploy action="redeploy" name="${app.name}"
                   deleteFiles="true" deltaFiles="${static.uri}"
                adminurl="${admin.url}" targets="${deploy.targets}"
                userconfigfile="${userconfig.file}" userkeyfile="$
{userkey.file}"
                failonerror="true" verbose="true"/>
```


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

```
    </target>
    <target name="deploy_library" depends="check-connection">
        <fail unless="library.source" message="Supply -Dlibrary.source=..."/>
        <wldeploy action="deploy" name="${library.name}" source="$
{library.source}"
                  library="true"
               adminurl="${admin.url}" targets="${deploy.targets}"
               userconfigfile="${userconfig.file}" userkeyfile="$
{userkey.file}"
               failonerror="true" verbose="true"/>
    </target>
</project>
```

The help target is informational, including for bare ant. Select a deployment/deletion target explicitly. The following host, port, target and absolute input paths are illustrative; use platform-appropriate command-line quoting.

```
ant -Dadmin.url=t3s://admin.example.com:7002 -Ddeploy.targets=myserver -
Duserconfig.file=/protected/deployer.config -Duserkey.file=/protected/
deployer.key -Dapp.source=/build/output/redeployEAR deploy
```

wldeploy Ant Task Attribute Reference

The following sections describe the attributes and child element `<files>` of the `wldeploy` Ant task.

Main Attributes

The following table describes the main attributes of the `wldeploy` Ant task.

These attributes mirror some of the arguments of the `weblogic.Deployer` command. Oracle provides an Ant task version of the `weblogic.Deployer` command so that developers can easily deploy and test their applications as part of the iterative development process. Typically, however, administrators use the `weblogic.Deployer` command, and not the `wldeploy` Ant task, to deploy applications in a production environment. For that reason, see the weblogic.Deployer Command-Line Reference in *Deploying Applications to Oracle WebLogic Server* for the full and complete definition of the attributes of the `wldeploy` Ant task. The table below is provided just as a quick summary.

Table B-1 Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
`action`	The deployment action to perform. Valid values are `deploy`, `cancel`, `undeploy`, `redeploy`, `distribute`, `start`, and `stop`.	String
`adminmode`	Specifies that the deployment action puts the application into Administration mode. Administration mode restricts access to an application to a configured Administration channel. Valid values for this attribute are `true` and `false`. Default value is `false`, which means that by default the application is deployed in production mode so that all clients can access it immediately.	Boolean

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

Table B-1 (Cont.) Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
`adminurl`	The URL of the Administration Server. The format of the value of this attribute is *`protocol://host:port`*, where *`protocol`* is either `http` or `t3`, *`host`* is the host on which the Administration Server is running, and *`port`* is the port which the Administration Server is listening. **Note:** In order to use the HTTP protocol, you must enable the HTTP tunnelling option in the WebLogic Remote Console.	String
`allversions`	Specifies that the action (redeploy, stop, and so on) applies to all versions of the application. Valid values for this attribute are `true` and `false`. The default value is `false`.	Boolean
`altappdd`	Specifies the name of an alternate Java EE deployment descriptor (`application.xml`) to use for deployment. If you do not specify this attribute, and you are deploying an enterprise application, the default deployment descriptor is called `application.xml` and is located in the META-INF subdirectory of the main application directory or archive (specified by the `source` attribute.)	String
`altwlsappdd`	Specifies the name of an alternate WebLogic Server deployment descriptor (`weblogic-application.xml`) to use for deployment. If you do not specify this attribute, and you are deploying an enterprise application, the default deployment descriptor is called `weblogic-application.xml` and is located in the META-INF subdirectory of the main application directory or archive (specified by the `source` attribute.)	String
`appversion`	The version identifier of the deployed application.	String
`debug`	Enable `wldeploy` debugging messages.	Boolean
`deleteFiles`	Specifies whether to remove static files from a server's staging directory. This attribute is valid only for unarchived deployments, and only for applications deployed using `stage` mode. You must specify target servers when using this attribute. Specifying the `deleteFiles` attributes indicates that WebLogic Server should remove only those files that it copied to the staging area during deployment. This attribute can be used only in combination with `action="redeploy"`. Because the `deleteFiles` attribute deletes all specified files, Oracle recommends that you use caution when using the `deleteFiles` attribute and that you do not use it in production environments. Valid values for this attribute are true and false. Default value is false.	Boolean
`deltaFiles`	Specifies a comma- or space-separated list of files, relative to the root directory of the application, which are to be redeployed. Use this attribute only in conjunction with `action="redeploy"` to perform a partial redeploy of an application.	String
`enableSecurityValidation`	Specifies whether or not to enable validation of security data. Valid values for this attribute are true and false. Default value is false.	Boolean
`externalStage`	Specifies whether the deployment uses `external_stage` deployment mode. In this mode, the Ant task does not copy the deployment files to target servers; instead, you must ensure that deployment files have been copied to the correct subdirectory in the target servers' staging directories. You can specify only one of the following attributes: `stage`, `nostage`, or `external_stage`. If none is specified, the default deployment mode to Managed Servers is `stage`; the default mode to the Administration Server and in single-server cases is `nostage`. See Controlling Deployment File Copying with Staging Modes.	Boolean

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

Table B-1 (Cont.) Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
`failonerror`	This is a global attribute used by WebLogic Server Ant tasks. It specifies whether the task should fail if it encounters an error during the build. Valid values for this attribute are true and false. Default value is true.	Boolean
`graceful`	Stops the application after existing HTTP clients have completed their work. You can use this attribute *only* when stopping or undeploying an application, or in other words, you must also specify either the `action="stop"` or `action="undeploy"` attributes. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
`id`	Identification used for obtaining status or cancelling the deployment. You assign a unique ID to an application when you deploy it, and then subsequently use the ID when redeploying, undeploying, stopping, and so on. If you do not specify this attribute, the Ant task assigns a unique ID to the application.	String
`ignoresessions`	This option immediately places the application into Administration mode without waiting for current HTTP sessions to complete. You can use this attribute *only* when stopping or undeploying an application, or in other words, you must also specify either the `action="stop"` or `action="undeploy"` attributes. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
`libImplVer`	Specifies the implementation version of a Java EE library or optional package. This attribute can be used only if the library or package does not include a implementation version in its manifest file. You can specify this attribute only in combination with the `library` attribute. See Creating Shared Java EE Libraries and Optional Packages.	String
`library`	Identifies the deployment as a shared Java EE library or optional package. You must specify the `library` attribute when deploying or distributing any Java EE library or optional package. Valid values for this attribute are `true` and `false`. Default value is `false`. See Creating Shared Java EE Libraries and Optional Packages.	Boolean
`libSpecVer`	Provides the specification version of a Java EE library or optional package. This attribute can be used only if the library or package does not include a specification version in its manifest file. You can specify this attribute only in combination with the `library` attribute. See Creating Shared Java EE Libraries and Optional Packages.	String
`name`	The deployment name for the deployed application. If you do not specify this attribute, WebLogic Server assigns a deployment name to the application, based on its archive file or exploded directory.	String
`nostage`	Specifies whether the deployment uses nostage deployment mode. In this mode, the Ant task does not copy the deployment files to target servers, but leaves them in a fixed location, specified by the `source` attribute. Target servers access the same copy of the deployment files. You can specify only one of the following attributes: `stage`, `nostage`, or `external_stage`. If none is specified, the default deployment mode to Managed Servers is `stage`; the default mode to the Administration Server and in single-server cases is `nostage`. See Controlling Deployment File Copying with Staging Modes.	Boolean
`noversion`	Indicates that the `wldeploy` Ant task should ignore all version related code paths on the Administration Server. This behavior is useful when deployment source files are located on Managed Servers (not the Administration Server) and you want to use the external_stage staging mode. If you use this option, you cannot use versioned applications. Valid values for this attribute are true and false. Default value is false.	Boolean

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

Table B-1 (Cont.) Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
`nowait`	Specifies whether `wldeploy` returns immediately after making a deployment call (by deploying as a background task).	Boolean
`password`	The administrative password. To avoid having the plain text password appear in the build file or in process utilities such as `ps`, first store a valid user name and encrypted password in a configuration file using the WebLogic Scripting Tool (WLST) `storeUserConfig` command. Then omit both the `username` and `password` attributes in your Ant build file. When the attributes are omitted, `wldeploy` attempts to login using values obtained from the default configuration file. If you want to obtain a user name and password from a non-default configuration file and key file, use the `userconfigfile` and `userkeyfile` attributes with `wldeploy`. See the command reference for `storeUserConfig` in the *WLST Command Reference for WebLogic Server* for more information on storing and encrypting passwords.	String
`plan`	Specifies a deployment plan to use when deploying the application or module. By default, `wldeploy` does not use an available deployment plan, even if you are deploying from an application root directory that contains a plan.	String
`planversion`	The version identifier of the deployment plan.	String
`remote`	Specifies whether the server is located on a different machine. This affects how filenames are transmitted. Valid values for this attribute are `true` and `false`. Default value is `false`, which means that the Ant task assumes that all source paths are valid paths on the local machine.	Boolean
`removePlanOverride`	Removes an overridden deployment plan during a redeploy or update deployment action. To remove an application override, specify the removePlanOverride attribute. You can specify the removePlanOverride attribute for the redeploy deployment actions.	String
`retiretimeout`	Specifies the number of seconds before WebLogic Server undeploys the currently-running version of this application or module so that clients can start using the new version. It is assumed, when you specify this attribute, that you are starting, deploying, or redeploying a new version of an already-running application. See Redeploying Applications in a Production Environment.	int
`securityModel`	Specifies the security model to use for this deployment. Possible security models are: • Deployment descriptors only • Customize roles • Customize roles and policies • Security realm configuration (advanced model) Valid actual values for this attribute are `DDOnly`, `CustomRoles`, `CustomRolesAndPolicy`, or `Advanced`. See Options for Securing Web application and EJB Resources for more information on these security models.	String
`source`	The archive file or exploded directory to deploy.	File
`stage`	Specifies whether the deployment uses stage deployment mode. In this mode, the Ant task copies deployment files to target servers' staging directories. You can specify only one of the following attributes: `stage`, `nostage`, or `external_stage`. If none is specified, the default deployment mode to Managed Servers is `stage`; the default mode to the Administration Server and in single-server cases is `nostage`. See Controlling Deployment File Copying with Staging Modes.	Boolean

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

Table B-1 (Cont.) Attributes of the wldeploy Ant Task

Attribute	Description	Data Type
`submoduletargets`	Specifies JMS server targets for resources defined within a JMS application module. The value of this attribute is a comma-separated list of JMS server names. See Using Sub-Module Targeting with JMS Application Modules.	String
`targets`	The list of target servers to which the application is deployed. The value of this attribute is a comma-separated list of the target servers, clusters, or virtual hosts. If you do not specify a target list when deploying an application, the target defaults to the Administration Server instance.	String
timeout	The maximum number of seconds to wait for a deployment to succeed.	int
`upload`	Specifies whether the source file(s) are copied to the Administration Server's upload directory prior to deployment. Use this attribute when you are on a remote machine and you cannot copy the deployment files to the Administration Server by other means. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
`usenonexclusivelock`	Specifies that the deployment action (deploy, redeploy, stop, and so on) uses the existing lock on the domain that has already been acquired by the same user performing the action. This attribute is particularly useful when the user is using multiple deployment tools (Ant task, command line, and so on) simultaneously and one of the tools has already acquired a lock on the domain. Valid values for this attribute are `true` and `false`. Default value is `false`.	Boolean
`user`	The administrative user name.	String
`userconfigfile`	Specifies the location of a user configuration file to use for obtaining the administrative user name and password. Use this option, instead of the `user` and `password` attributes, in your build file when you do not want to have the plain text password shown in-line or in process-level utilities such as `ps`. Before specifying the `userconfigfile` attribute, you must first generate the file using the WebLogic Scripting Tool (WLST) `storeUserConfig` command as described in the *WLST Command Reference for Oracle WebLogic Server*.	String
`userkeyfile`	Specifies the location of a user key file to use for encrypting and decrypting the user name and password information stored in a user configuration file (the `userconfigfile` attribute). Before specifying the `userkeyfile` attribute, you must first generate the key file using the WebLogic Scripting Tool (WLST) `storeUserConfig` command as described in the *WLST Command Reference for Oracle WebLogic Server*.	String
`verbose`	Specifies whether `wldeploy` displays verbose output messages.	Boolean

Nested <files> Child Element

The `wldeploy` Ant task also includes the `<files>` child element that can be nested to specify a list of files on which to perform a deployment action (for example, a list of JSPs to undeploy.)

> **Note**
>
> Use of `<files>` to redeploy a list of files in an application has been deprecated as of release 9.0 of WebLogic Server. Instead, use the `deltaFiles` attribute of wldeploy.

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

The <files> element works the same as the standard <fileset> Ant task (except for the difference in actual task name). Therefore, see the Apache Ant Web site at http://ant.apache.org/manual/Types/fileset.html for detailed reference information about the attributes you can specify for the <files> element.

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