Oracle® Fusion Middleware

Developing Applications with the WebLogic Security Service



26 (26.1.0.0.0)
G56942-01
November 2026

ORACLE

Oracle Fusion Middleware Developing Applications with the WebLogic Security Service, 26 (26.1.0.0.0)

G56942-01

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 ii
Diversity and Inclusion ii
Related Information ii
Conventions iv

1 WebLogic Security Programming Overview

What Is Security? 1
WebLogic Remote Console and Security 2
Types of Security Supported by WebLogic Server 2
- Authentication 2
- Authorization 2
- Jakarta Security 3

Security APIs 3
- JAAS Client Application APIs 3
  - Java JAAS Client Application APIs 3
  - WebLogic JAAS Client Application APIs 4
- SSL Client Application APIs 4
  - Java SSL Client Application APIs 4
  - WebLogic SSL Client Application APIs 4
- Other APIs 5

2 Securing Web Applications

Authentication With Web Browsers 1
- User Name and Password Authentication 2
- Digital Certificate Authentication 3

Multiple Web Applications, Cookies, and Authentication 5
- Using Secure Cookies to Prevent Session Stealing 5
  - Configuring the Session Cookie as a Secure Cookie 5
  - Using the AuthCookie _WL_AUTHCOOKIE_JSESSIONID 6

Developing Secure Web Applications 7

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

Developing BASIC Authentication Web Applications 7
Using HttpSessionListener to Account for Browser Caching of Credentials 11
Understanding BASIC Authentication with Unsecured Resources 12
Setting the enforce-valid-basic-auth-credentials Flag 12
Check the Value of enforce-valid-basic-auth-credentials 12
Developing FORM Authentication Web Applications 13
Using Identity Assertion for Web Application Authentication 18
Using Two-Way SSL for Web Application Authentication 19
Providing a Fallback Mechanism for Authentication Methods 19
Configuration 20
Developing Swing-Based Authentication Web Applications 20
Deploying Web Applications 20
Using Declarative Security With Web Applications 21
Web Application Security-Related Deployment Descriptors 22
web.xml Deployment Descriptors 22
auth-constraint 22
security-constraint 23
security-role 25
security-role-ref 25
user-data-constraint 26
web-resource-collection 27
weblogic.xml Deployment Descriptors 27
externally-defined 28
run-as-principal-name 29
run-as-role-assignment 30
security-permission 30
security-permission-spec 31
security-role-assignment 31
Using Programmatic Security With Web Applications 32
Jakarta Security SecurityContext Methods 32
Servlet HttpServletRequest Methods 33
getUserPrincipal 33
isUserInRole 33
Authenticating Users Programmatically 35
Using the Jakarta Security SecurityContext Interface 35
Using the Programmatic Authentication API 35
Change the User's Session ID at Login 36

3 Using JAAS Authentication in Java Clients

JAAS and WebLogic Server 1
JAAS Authentication Development Environment 2

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

JAAS Authentication APIs 3
JAAS Client Application Components 5
WebLogic LoginModule Implementation 7
JVM-Wide Default User and the runAs() Method 7
Writing a Client Application Using JAAS Authentication 8
Using JNDI Authentication 11
Java Client JAAS Authentication Code Examples 12

4 Using SSL Authentication in Java Clients

JSSE and WebLogic Server 1
Using JNDI Authentication 2
SSL Certificate Authentication Development Environment 4
SSL Authentication APIs 4
SSL Client Application Components 6
Writing Applications that Use SSL 7
Communicating Securely From WebLogic Server to Other WebLogic Servers 8
Writing SSL Clients 8
SSLClient Sample 8
SSLSocketClient Sample 9
Using Two-Way SSL Authentication 10
Two-Way SSL Authentication with JNDI 11
Writing a User Name Mapper 14
Using Two-Way SSL Authentication Between WebLogic Server Instances 15
Using Two-Way SSL Authentication with Servlets 16
Using a Custom Host Name Verifier 17
Using a Trust Manager 18
Using the CertPath Trust Manager 20
Using a Handshake Completed Listener 20
Using an SSLContext 21
Using URLs to Make Outbound SSL Connections 21
SSL Client Code Examples 23

5 Securing EJBs

Jakarta EE Architecture Security Model 1
Declarative Security 1
Declarative Authorization Via Annotations 2
Programmatic Security 2
Declarative Versus Programmatic Authorization 3
Using Declarative Security With EJBs 3
Implementing Declarative Security Via Metadata Annotations 3

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

Security-Related Annotation Code Examples 4
Implementing Declarative Security Via Deployment Descriptors 4
EJB Security-Related Deployment Descriptors 6
ejb-jar.xml Deployment Descriptors 6
method 6
method-permission 7
role-name 8
run-as 8
security-identity 8
security-role 9
security-role-ref 9
unchecked 10
use-caller-identity 10
weblogic-ejb-jar.xml Deployment Descriptors 11
client-authentication 11
client-cert-authentication 12
confidentiality 12
externally-defined 12
identity-assertion 14
iiop-security-descriptor 15
integrity 15
principal-name 16
role-name 16
run-as-identity-principal 16
run-as-principal-name 18
run-as-role-assignment 18
security-permission 20
security-permission-spec 20
security-role-assignment 21
transport-requirements 21
Using Programmatic Security With EJBs 21
SecurityContext Interface Methods 22
EJBContext Interface Methods 22

6 Using Network Connection Filters

The Benefits of Using Network Connection Filters 1
Network Connection Filter API 1
Connection Filter Interfaces 2
ConnectionFilter Interface 2
ConnectionFilterRulesListener Interface 2
Connection Filter Classes 3

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

ConnectionFilterImpl Class 3
ConnectionEvent Class 3
Guidelines for Writing Connection Filter Rules 3
Connection Filter Rules Syntax 3
Types of Connection Filter Rules 4
How Connection Filter Rules are Evaluated 5
Configuring the WebLogic Connection Filter 5
Developing Custom Connection Filters 5

7 Using Java Security Features to Protect WebLogic Resources

Using Jakarta Security to Protect WebLogic Resources 1
Using the Java Security Manager to Protect WebLogic Resources 2
Setting Up the Java Security Manager 2
Modifying your Custom Policy File for General Use 3
Setting Application-Type Security Policies 4
Setting Application-Specific Security Policies 5
Using Printing Security Manager 5
Printing Security Manager Startup Arguments 6
Starting WebLogic Server With Printing Security Manager 6
Writing Output Files 7
Using Jakarta Authorization 7
Comparing the WebLogic JACC Provider with the WebLogic Authorization Provider 8
Enabling the WebLogic JACC Provider 9

8 SAML APIs

SAML API Description 1
Configuring SAML SSO Attribute Support 2
What Are SAML SSO Attributes? 3
APIs for SAML Attributes 3
SAML 2.0 Basic Attribute Profile Required 3
Passing Multiple Attributes to SAML Credential Mappers 4
How to Implement SAML Attributes 4
Examples of the SAML 2.0 Attribute Interfaces 6
Example Custom SAML 2.0 Credential Attribute Mapper 6
Custom SAML 2.0 Identity Asserter Attribute Mapper 8
Make the Custom SAML Credential Attribute Mapper Class Available in the Console 10
Make the Custom SAML Identity Asserter Class Available in the Console 11

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

9 Using CertPath Building and Validation

CertPath Building 1
Instantiate a CertPathSelector 1
Instantiate a CertPathBuilderParameters 2
Use the JDK CertPathBuilder Interface 3
Example Code Flow for Looking Up a Certificate Chain 3
CertPath Validation 4
Instantiate a CertPathValidatorParameters 4
Use the JDK CertPathValidator Interface 5
Example Code Flow for Validating a Certificate Chain 6

10 Using Jakarta Authentication for a Web Application

Overview of Jakarta Authentication 1
Do You Need to Implement an Authentication Configuration Provider? 2
Do You Need to Implement a Principal Validation Provider? 2
Implement a SAM 3
Configure Jakarta Authentication for the Deployed Web Application 3

11 Using Jakarta Security

Overview of Jakarta Security in WebLogic Server 1
About the HttpAuthenticationMechanism Interface 2
HttpAuthenticationMechanism Interface Methods 3
HttpAuthenticationMechanism Interface Annotations 3
About the Identity Store Interfaces 3
IdentityStore Interface 4
IdentityStoreHandler 4
IdentityStore Interface Methods 4
RememberMeIdentityStore Interface 5
Usage Requirements 6

A Deprecated Security APIs

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

Preface

This document explains how to use the WebLogic Server security programming features.

Audience

This document is intended for the following audiences:

- Application Developers

  Java programmers who focus on developing client applications, adding security to Web applications and EJBs. They work with other engineering, Quality Assurance (QA), and database teams to implement security features. Application developers have in-depth/working knowledge of Java (including Jakarta Platform, Enterprise Edition (Jakarta EE) components such as servlets/JSPs and JSSE) and Java security.

  Application developers use the WebLogic security and Java security application programming interfaces (APIs) to secure their applications. Therefore, this document provides instructions for using those APIs for securing Web applications, Java applications, and EJBs.

- Security Developers

  Developers who focus on defining the system architecture and infrastructure for security products that integrate into WebLogic Server and on developing custom security providers for use with WebLogic Server. They work with application architects to ensure that the security architecture is implemented according to design and that no security holes are introduced. They also work with WebLogic Server administrators to ensure that security is properly configured. Security developers have a solid understanding of security concepts, including authentication, authorization, auditing (AAA), in-depth knowledge of Java (including Java Management eXtensions (JMX), and working knowledge of WebLogic Server and security provider functionality.

  Security developers use the Security Service Provider Interfaces (SSPIs) to develop custom security providers for use with WebLogic Server. This document does not address this task; for information on how to use the SSPIs to develop custom security providers, see Overview of the Development Process in *Developing Security Providers for Oracle WebLogic Server*.

- Server Administrators

  Administrators who work closely with application architects to design a security scheme for the server and the applications running on the server, to identify potential security risks, and to propose configurations that prevent security problems. Related responsibilities may include maintaining critical production systems, configuring and managing security realms, implementing authentication and authorization schemes for server and application resources, upgrading security features, and maintaining security provider databases. WebLogic Server administrators have in-depth knowledge of the Java security architecture, including Web application and EJB security, Public Key security, and SSL.

- Application Administrators

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

Administrators who work with WebLogic Server administrators to implement and maintain security configurations and authentication and authorization schemes, and to set up and maintain access to deployed application resources in defined security realms. Application administrators have general knowledge of security concepts and the Java Security architecture. They understand Java, XML, deployment descriptors, and can identify security events in server and audit logs.

While administrators typically use WebLogic Remote Console to deploy, configure, and manage applications when they put the applications into production, application developers may also use WebLogic Remote Console to test their applications before they are put into production. At a minimum, testing requires that applications be deployed and configured. This document does not cover some aspects of administration as it relates to security, rather, it references *Administering Security for Oracle WebLogic Server*, *Securing Resources Using Roles and Policies for Oracle WebLogic Server*, and *Oracle WebLogic Remote Console Online Help* for descriptions of how to use WebLogic Remote Console to perform security tasks.

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 Information

In addition to this document, *Developing Applications with the WebLogic Security Service*, the following documents provide information on the WebLogic Security Service:

- *Understanding Security for Oracle WebLogic Server*—This document summarizes the features of the WebLogic Security Service and presents an overview of the architecture and capabilities of the WebLogic Security Service. It is the starting point for understanding the WebLogic Security Service.
- *Securing a Production Environment for Oracle WebLogic Server*— This document highlights essential security measures for you to consider before you deploy WebLogic Server into a production environment.

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

- *Developing Security Providers for Oracle WebLogic Server*—This document provides security vendors and application developers with the information needed to develop custom security providers that can be used with WebLogic Server.
- *Administering Security for Oracle WebLogic Server*—This document explains how to configure security for WebLogic Server.
- *Securing Resources Using Roles and Policies for Oracle WebLogic Server*—This document introduces the various types of WebLogic resources, and provides information that allows you to secure these resources using WebLogic Server.
- *Oracle WebLogic Remote Console Online Help*—This document describes how to use the WebLogic Remote Console to perform security tasks.
- *Java API Reference for Oracle WebLogic Server* —This document includes reference documentation for the WebLogic security packages that are provided with and supported by the WebLogic Server software.

Security Samples and Tutorials

In addition to the documents listed in Related Information, Oracle provides a rich set of code examples and sample applications that show several approaches to learning about and working with WebLogic Server. These examples and sample applications are available through a separate WebLogic Server examples installer.

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

Security Examples in the WebLogic Server Distribution

The following examples illustrate WebLogic Server security features:

- Java Authentication and Authorization Service (JAAS)
- SAML 2.0 For Web SSO Scenario
- Outbound and Two-way SSL

The WebLogic Server installation also includes an example demonstrating the use of the built-in database identity store functionality provided by the Jakarta Security specification. This example is located in the *EXAMPLES_HOME*`\examples\src\examples\javaee8\security` directory.

The security tasks and code examples provided in this document assume that you are using the WebLogic security providers that are included in the WebLogic Server distribution, not custom security providers. The usage of the WebLogic security APIs does not change if you elect to use custom security providers, however, the management procedures of your custom security providers may be different.

> **Note**
>
> This document does not provide comprehensive instructions on how to configure WebLogic Security providers or custom security providers. For information on configuring WebLogic security providers and custom security providers, see Configuring Security Providers in *Administering Security for Oracle WebLogic Server*.

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

New and Changed WebLogic Server Features

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

Conventions

The following text conventions are used in this document:

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

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

1

WebLogic Security Programming Overview

Oracle WebLogic Server supports the ability to incorporate standard Java security technologies such as the Java Authentication and Authorization Service (JAAS), Java Secure Sockets Extensions (JSSE), Java Cryptography Architecture and Java Cryptography Extensions (JCE), the Jakarta Authentication, and Jakarta Security specifications in hosted applications, such as web applications, web services, EJB, and more, and includes support for implementing declarative and programmatic authorization in those applications.

- What Is Security?
- WebLogic Remote Console and Security
- Types of Security Supported by WebLogic Server
- Security APIs

What Is Security?

Security refers to techniques for ensuring that data stored in a computer or passed between computers is not compromised.Most security measures involve proof material and data encryption. Proof material is typically a secret word or phrase that gives a user access to a particular application or system. Data encryption is the translation of data into a form that cannot be interpreted without holding or supplying the same secret.
Distributed applications, such as those used for electronic commerce (e-commerce), offer many access points at which malicious people can intercept data, disrupt operations, or generate fraudulent input. As a business becomes more distributed the probability of security breaches increases. Accordingly, as a business distributes its applications, it becomes increasingly important for the distributed computing software upon which such applications are built to provide security.

An application server resides in the sensitive layer between end users and your valuable data and resources. Oracle WebLogic Server provides authentication, authorization, and encryption services with which you can guard these resources. These services cannot provide protection, however, from an intruder who gains access by discovering and exploiting a weakness in your deployment environment.

Therefore, whether you deploy WebLogic Server on the Internet or on an intranet, it is a good idea to hire an independent security expert to go over your security plan and procedures, audit your installed systems, and recommend improvements.

Another good strategy is to read as much as possible about security issues and appropriate security measures. The document *Securing a Production Environment for Oracle WebLogic Server*highlights essential security measures for you to consider before you deploy WebLogic Server into a production environment. The document *Securing Resources Using Roles and Policies for Oracle WebLogic Server*introduces the various types of WebLogic resources, and provides information that allows you to secure these resources using WebLogic Server. For the latest information about securing Web servers, Oracle also recommends reading the Security Improvement Modules, Security Practices, and Technical Implementations information (`http://www.cert.org/`) available from the CERT™ Coordination Center operated by Carnegie Mellon University.

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

Oracle suggests that you apply the remedies recommended in our security advisories. In the event of a problem with an Oracle product, Oracle distributes an advisory and instructions with the appropriate course of action. If you are responsible for security related issues at your site, please register to receive future notifications.

WebLogic Remote Console and Security

You can use the WebLogic Remote Console to define and edit deployment descriptors for Web Applications, EJBs, Jakarta Connectors, and Enterprise Applications.This document, *Developing Applications with the WebLogic Security Service*, does not describe how to use the WebLogic Remote Console to configure security. For information on how to use the WebLogic Remote Console to define and edit deployment descriptors, see *Securing Resources Using Roles and Policies for Oracle WebLogic Server* and *Administering Security for Oracle WebLogic Server*.

Types of Security Supported by WebLogic Server

WebLogic Server supports security mechanisms such as authentication, authorization, and Jakarta EE security in deployed applications.

- Authentication
- Authorization
- Jakarta Security

Authentication

Authentication is the mechanism by which callers and service providers prove that they are acting on behalf of specific users or systems. Authentication answers the question, "Who are you?" using credentials. When the proof is bidirectional, it is referred to as mutual authentication.

WebLogic Server supports username and password authentication and certificate authentication. For certificate authentication, WebLogic Server supports both one-way and two-way SSL (Secure Sockets Layer) authentication. Two-way SSL authentication is a form of mutual authentication.

In WebLogic Server, Authentication providers are used to prove the identity of users or system processes. Authentication providers also remember, transport, and make identity information available to various components of a system (via subjects) when needed. You can configure the Authentication providers using the Web application and EJB deployment descriptor files, or WebLogic Remote Console, or a combination of both.

Authorization

Authorization is the process whereby the interactions between users and WebLogic resources are controlled, based on user identity or other information. In other words, authorization answers the question, "What can you access?"

In WebLogic Server, a WebLogic Authorization provider is used to limit the interactions between users and WebLogic resources to ensure integrity, confidentiality, and availability. You can configure the Authorization provider using the Web application and EJB deployment descriptor files, or WebLogic Remote Console, or a combination of both.

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

WebLogic Server also supports the use of programmatic authorization (also referred to in this document as programmatic security) to limit the interactions between users and WebLogic resources.

Jakarta Security

For implementation and use of user authentication and authorization, WebLogic Server utilizes the security services of the JDK and Jakarta EE components. These security services are based on standardized, modular components. WebLogic Server implements these Java security service methods according to the standard, and adds extensions that handle many details of application behavior automatically, without requiring additional programming.

WebLogic Server supports the Jakarta Security specification (`https://jakarta.ee/specifications/security/`) , which defines portable, plug-in interfaces for HTTP authentication and identity stores, and an injectable `SecurityContext` interface that provides an API for programmatic security. You can use the built-in implementations of the plug-in SPIs, or write custom implementations.

Security APIs

WebLogic Server supports and implements several security packages and classes. You use these packages to secure interactions between WebLogic Server and client applications, EJBs, and Web applications.

The following topics are covered in this section:

- JAAS Client Application APIs
- SSL Client Application APIs
- Other APIs

> **Note**
>
> Several of the WebLogic security packages, classes, and methods are deprecated in this release of WebLogic Server. For more detailed information on deprecated packages and classes, see Deprecated Security APIs.

JAAS Client Application APIs

You use Java APIs and WebLogic APIs to write client applications that use JAAS authentication.

The following topics are covered in this section:

- Java JAAS Client Application APIs
- WebLogic JAAS Client Application APIs

Java JAAS Client Application APIs

You use the following Java APIs to write JAAS client applications. The APIs are available at Java SE and JDK API Specification .

- javax.naming

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

- javax.security.auth
- javax.security.auth.callback
- javax.security.auth.login
- javax.security.auth.spi

For information on how to use these APIs, see JAAS Authentication APIs.

WebLogic JAAS Client Application APIs

You use the following WebLogic APIs to write JAAS client applications:

- weblogic.security
- weblogic.security.auth
- weblogic.security.auth.callback

For information on how to use these APIs, see JAAS Authentication APIs.

SSL Client Application APIs

You use Java and WebLogic APIs to write client applications that use SSL authentication:

The following topics are covered in this section:

- Java SSL Client Application APIs
- WebLogic SSL Client Application APIs

Java SSL Client Application APIs

You use the following Java APIs (available from Java SE and JDK API Specification and Jakarta EE Platform API) to write SSL client applications:

- java.security
- java.security.cert
- javax.crypto
- javax.naming
- javax.net
- javax.security
- jakarta.servlet
- jakarta.servet.http

WebLogic Server also supports the javax.net.SSL API (see Java SE and JDK API Specification), but Oracle recommends that you use the `weblogic.security.SSL` package when you use SSL with WebLogic Server.

For information on how to use these APIs, see SSL Authentication APIs.

WebLogic SSL Client Application APIs

You use the following WebLogic APIs to write SSL client applications.

- weblogic.net.http

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

- weblogic.security.SSL

For information on how to use these APIs, see SSL Authentication APIs.

Other APIs

Additionally, you use the following APIs to develop WebLogic Server applications:

- `weblogic.security.jacc`

  This API provides the `RoleMapper` interface. If you implement Jakarta Authorization (formerly JACC), you can use this package with the `jakarta.security.jacc` package. For information about the WebLogic JACC provider, see Using Jakarta Authorization. For information about developing a Jakarta Authorization provider, see the `jakarta.security.jacc` package Javadoc at `https://jakarta.ee/specifications/platform/11/apidocs/jakarta/security/jacc/package-summary`.

- `weblogic.security.net`

  This API provides interfaces and classes that are used to implement network connection filters. Network connection filters allow or deny connections to Oracle WebLogic Server based on attributes such as the IP address, domain, or protocol of the initiator of the network connection. For more information about how to use this API, see Using Network Connection Filters.

- `weblogic.security.pk`

  This API provides interfaces and classes to build and validate certification paths. See Using CertPath Building and Validation for information on using this API to build and validate certificate chains.

  See the `java.security.cert` package in Java SE and JDK API Specification for additional information on certificates and certificate paths.

- `com.bea.security.saml2.providers`

  This API provides interfaces and classes that are used to perform mapping of user and group information to Security Assertion Markup Language (SAML) assertions, and to cache and retrieve SAML assertions.

  SAML is an XML-based framework for exchanging security information. WebLogic Server supports SAML v2.0, including the Browser/Post and Browser/Artifact profiles. SAML authorization is not supported.

  For more information about SAML, see `http://www.oasis-open.org`.

- `weblogic.security.service`

  This API includes interfaces, classes, and exceptions that support security providers. The WebLogic Security Framework consists of interfaces, classes, and exceptions provided by this API. The interfaces, classes, and exceptions in this API should be used in conjunction with those in the `weblogic.security.spi` package. For more information about how to use this API, see Security Providers and WebLogic Resources in *Developing Security Providers for Oracle WebLogic Server*.

- `weblogic.security.services`

  This API provides the server-side authentication class. This class is used to perform a local login to the server. It provides login methods that are used with CallbackHandlers to authenticate the user and return credentials using the default security realm.

- `weblogic.security.spi`

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

This package provides the Security Service Provider Interfaces (SSPIs). It provides interfaces, classes, and exceptions that are used for developing custom security providers. In many cases, these interfaces, classes, and exceptions should be used in conjunction with those in the `weblogic.security.service` API. You implement interfaces, classes, and exceptions from this package to create runtime classes for security providers. For more information about how to use the SSPIs, see Security Services Provider Interfaces (SSPIs) in *Developing Security Providers for Oracle WebLogic Server*.

- `weblogic.servlet.security`

  This API provides a server-side API that supports programmatic authentication from within a servlet application. For more about how to use this API, see Using the Programmatic Authentication API.

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

2

Securing Web Applications

Oracle WebLogic Server supports the Jakarta EE architecture security model for securing Web applications, which includes support for declarative authorization (also referred to as declarative security) and programmatic authorization (also referred to as programmatic security).

- Authentication With Web Browsers
- Multiple Web Applications, Cookies, and Authentication
- Developing Secure Web Applications
- Using Declarative Security With Web Applications
- Web Application Security-Related Deployment Descriptors
- Using Programmatic Security With Web Applications
- Using the Programmatic Authentication API

> **Note**
>
> You can use deployment descriptor files and WebLogic Remote Console to secure Web applications. This document describes how to use deployment descriptor files. For information on using WebLogic Remote Console to secure Web applications, see *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

To implement programmatic authorization in Web applications, WebLogic Server supports the use of:

- The Servlet `HttpServletRequest.isUserInRole` and `HttpServletRequest.getUserPrincipal` methods
- The `security-role-ref` element in deployment descriptors
- The Jakarta Security `SecurityContext.getCallerPrincipal`, `SecurityContext.getPrincipalsByType`, `SecurityContext.isCallerInRole`, and `SecurityContext.hasAccessToWebResource` methods

Authentication With Web Browsers

Web browsers can connect to WebLogic Server over either a HyperText Transfer Protocol (HTTP) port or an HTTP with SSL (HTTPS) port. WebLogic Server uses encryption and digital certificate authentication when Web browsers connect to the server using the HTTPS port.

The benefits of using an HTTPS port versus an HTTP port are two-fold. With HTTPS connections:

- All communication on the network between the Web browser and the server is encrypted. None of the communication, including the user name and password, is in clear text.

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

- As a minimum authentication requirement, the server is required to present a digital certificate to the Web browser client to prove its identity.

If the server is configured for two-way SSL authentication, both the server and client are required to present a digital certificate to each other to prove their identity.

User Name and Password Authentication

WebLogic Server performs user name and password authentication when users use a Web browser to connect to the server via the HTTP port. In this scenario, the browser and an instance of WebLogic Server interact in the following manner to authenticate a user (see Figure 2-1):

1. A user invokes a WebLogic resource in Oracle WebLogic Server by entering the URL for that resource in a Web browser. The HTTP URL contains the HTTP listen port, for example, `http://myserver:7001`.
2. The Web server in Oracle WebLogic Server receives the request.

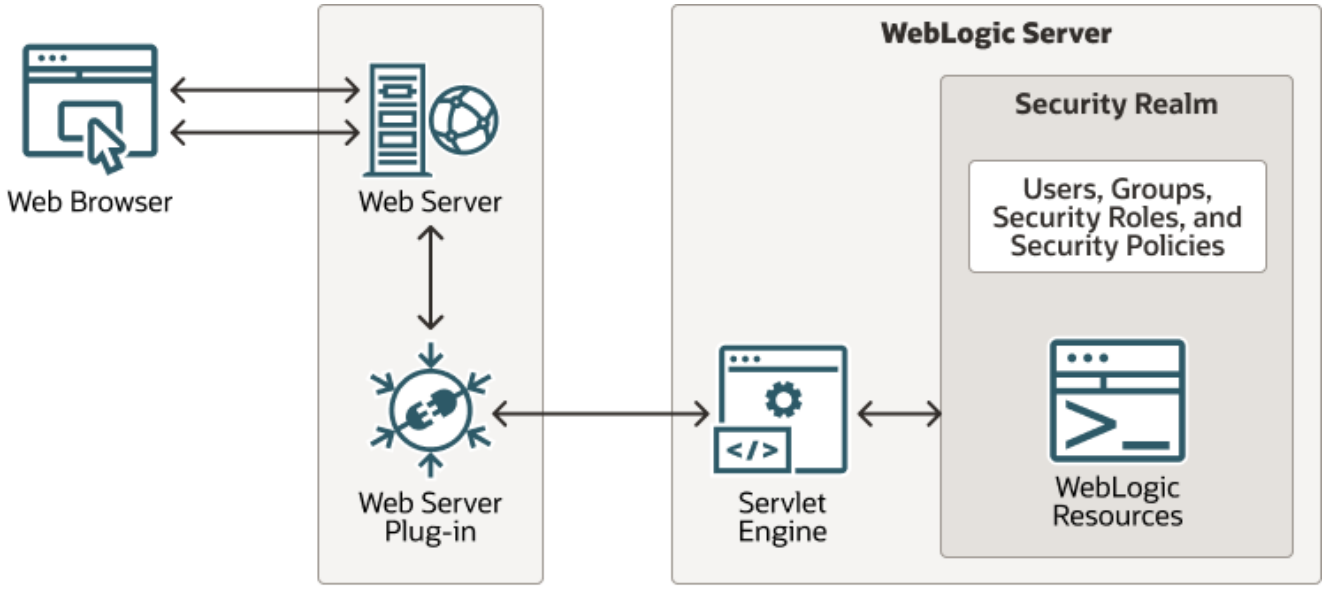
   > **Note**
   >
   > Oracle WebLogic Server provides its own Web server but also supports the use of Apache Server, Microsoft Internet Information Server, and Java System Web Server as Web servers.

3. The Web server determines whether the WebLogic resource is protected by a security policy. If the WebLogic resource is protected, the Web server uses the established HTTP connection to request a user name and password from the user.
4. When the user's Web browser receives the request from the Web server, it prompts the user for a user name and password.
5. The Web browser sends the request to the Web server again, along with the user name and password.
6. The Web server forwards the request to the Web server plug-in. Oracle WebLogic Server provides the following plug-ins for Web servers:
   - Apache-WebLogic Server plug-in
   - Java System Web Server plug-in
   - Internet Information Server (IIS) plug-in

   The Web server plug-in performs authentication by sending the request, via the HTTP protocol, to Oracle WebLogic Server, along with the authentication data (user name and password) received from the user.
7. Upon successful authentication, Oracle WebLogic Server proceeds to determine whether the user is authorized to access the WebLogic resource.
8. Before invoking a method on the WebLogic resource, the WebLogic Server instance performs a security authorization check. During this check, the server security extracts the user's credentials from the security context, determines the user's security role, compares the user's security role to the security policy for the requested WebLogic resource, and verifies that the user is authorized to invoke the method on the WebLogic resource.
9. If authorization succeeds, the server fulfills the request.

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

Figure 2-1 Secure Login for Web Browsers



Note: Username/Password authentication can be required for HTTP and one-way SSL authentication. HTTPS connections can be configured for one-way or two-way SSL authentication.

Digital Certificate Authentication

WebLogic Server uses encryption and digital certificate authentication when Web browser users connect to the server via the HTTPS port. In this scenario, the browser and WebLogic Server instance interact in the following manner to authenticate and authorize a user (see Figure 2-1):

1. A user invokes a WebLogic resource in Oracle WebLogic Server by entering the URL for that resource in a Web browser. The HTTPS URL contains the SSL listen port, for example, `https://myserver:7002`.
2. The Web server in Oracle WebLogic Server receives the request.

> **Note**
>
> Oracle WebLogic Server provides its own Web server but also supports the use of Apache Server, Microsoft Internet Information Server, and Java System Web Server as Web servers.

3. The Web server checks whether the WebLogic resource is protected by a security policy. If the WebLogic resource is protected, the Web server uses the established HTTPS connection to request a user name and password from the user.
4. When the user's Web browser receives the request from Oracle WebLogic Server, it prompts the user for a user name and password. (This step is optional.)
5. The Web browser sends the request again, along with the user name and password. (Only supplied if requested by the server.)
6. WebLogic Server presents its digital certificate to the Web browser.

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

7. The Web browser checks that the server's name used in the URL (for example, `myserver`) matches the name in the digital certificate and that the digital certificate was issued by a trusted third party, that is, a trusted CA
8. If two-way SSL authentication is in force on the server, the server requests a digital certificate from the client.

> **Note**
>
> Even though WebLogic Server cannot be configured to enforce the full two-way SSL handshake with 1.0 Web Server proxy plug-ins, proxy plug-ins can be configured to provide the client certificate to the server if it is needed. To do this, configure the proxy plug-in to export the client certificate in the HTTP Header for WebLogic Server. For instructions on how to configure proxy plug-ins to export the client certificate to WebLogic Server, see the configuration information for the specific plug-in in *Using Oracle WebLogic Server Proxy Plug-Ins*.
>
> The latest plug-ins provide two-way SSL support for verifying client identity. Two-way SSL is automatically enforced when WebLogic Server requests the client certificate during the handshake process. See Configuring Two-Way SSL Between the Plug-In and Oracle WebLogic Server in *Using Oracle WebLogic Server Proxy Plug-Ins*.

9. The Web server forwards the request to the Web server plug-in. If secure proxy is set (this is the case if the HTTPS protocol is being used), the Web server plug-in also performs authentication by sending the request, via the HTTPS protocol, to the WebLogic resource in Oracle WebLogic Server, along with the authentication data (user name and password) received from the user.

> **Note**
>
> When using two-way SSL authentication, you can also configure the server to do identity assertion based on the client's certificate, where, instead of supplying a user name and password, the server extracts the user name and password from the client's certificate.

10. Upon successful authentication, Oracle WebLogic Server proceeds to determine whether the user is authorized to access the WebLogic resource.
11. Before invoking a method on the WebLogic resource, the server performs a security authorization check. During this check, the server extracts the user's credentials from the security context, determines the user's security role, compares the user's security role to the security policy for the requested WebLogic resource, and verifies that the user is authorized to invoke the method on the WebLogic resource.
12. If authorization succeeds, the server fulfills the request.

See the following topics:

- Configuring SSL
- Installing and Configuring the Apache HTTP Server Plug-In
- Installing and Configuring the Microsoft IIS Plug-In

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

Multiple Web Applications, Cookies, and Authentication

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

If you want to require separate authentication for a Web application, you can specify a unique cookie name or cookie path for the Web application. Specify the cookie name using the `CookieName` parameter and the cookie path with the `CookiePath` parameter, defined in the `weblogic.xml` `<session-descriptor>` element. See session-descriptor in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

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

However, note that a common Web security problem is session stealing. WebLogic Server provides two features, or methods, that Web site designers can use to prevent session stealing, described in Using Secure Cookies to Prevent Session Stealing.

Using Secure Cookies to Prevent Session Stealing

Session stealing happens when an attacker manages to get a copy of your session cookie, generally while the cookie is being transmitted over the network. This can only occur when the data is being sent in clear-text; that is, the cookie is not encrypted. WebLogic Server provides two features for securing session cookies.

- Configuring the Session Cookie as a Secure Cookie
- Using the AuthCookie _WL_AUTHCOOKIE_JSESSIONID

> **Note**
>
> These two features work correctly when the SSL request is terminated at WebLogic Server. Proxy architectures that terminate the SSL connection at a Web server plug-in or hardware load balancer can enable the `WeblogicPluginEnabled` attribute for these features to work, but doing so exposes the session cookie behind the proxy.

Configuring the Session Cookie as a Secure Cookie

You can prevent session stealing by configuring the application to use HTTPS. When communication with WebLogic Server is secured by SSL, you can have WebLogic Server make the session cookie secure by specifying the `<cookie-secure>` element in the `weblogic.xml` deployment descriptor and setting its value to `true`. A secure cookie indicates to the Web browser that the cookie should be sent using only a secure protocol, such as SSL.

Note that it is possible for an application with code running in the browser — for example, an applet — to make non-HTTP outbound connections. In such connections, the browser sends the session cookie. However, by specifying the `<cookie-http-only>` element in `weblogic.xml`, you constrain the browser to send the cookie only over an HTTP connection — the cookie is made inaccessible to applications or other protocols running in the browser. So if you specify

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

<cookie-http-only> in conjunction with <cookie-secure>, you ensure that session cookies are sent only over HTTPS.

For more information about the <cookie-secure> and <cookie-http-only> elements, see weblogic.xml Deployment Descriptor Elements in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Using the AuthCookie _WL_AUTHCOOKIE_JSESSIONID

WebLogic Server allows a user to securely access HTTPS resources in a session that was initiated using HTTP, without loss of session data. To enable this feature, ensure that `WebAppContainerMBean.AuthCookieEnabled` is set to `true`.

`AuthCookieEnabled` is enabled by default. If it is disabled, you can use WebLogic Remote Console to re-enable it:

1. In the **Edit Tree**, go to **Environment**, then **Domain**.
2. On the **Web Application** tab, turn on the **Auth Cookie Enabled** option.
3. Save and commit your changes.

When `AuthCookieEnabled` is set to `true`, the WebLogic Server instance sends a new secure cookie, `_WL_AUTHCOOKIE_JSESSIONID`, to the browser when authenticating via an HTTPS connection. Once the secure cookie is set, the session is allowed to access other security-constrained HTTPS resources only if the cookie is sent from the browser.

Thus, WebLogic Server uses two cookies: the `JSESSIONID` cookie and the `_WL_AUTHCOOKIE_JSESSIONID` cookie. By default, the `JSESSIONID` cookie is never secure, but the `_WL_AUTHCOOKIE_JSESSIONID` cookie is always secure. A secure cookie is only sent when an encrypted communication channel is in use. Assuming a standard HTTPS login (HTTPS is an encrypted HTTP connection), your browser gets both cookies.

For subsequent HTTP access, you are considered authenticated if you have a valid `JSESSIONID` cookie, but for HTTPS access, you must have both cookies to be considered authenticated. If you only have the `JSESSIONID` cookie, you must re-authenticate.

With this feature enabled, once you have logged in over HTTPS, the secure cookie is only sent encrypted over the network and therefore can never be stolen in transit. The `JSESSIONID` cookie is still subject to in-transit hijacking. Therefore, a Web site designer can ensure that session stealing is not a problem by making all sensitive data require HTTPS. While the HTTP session cookie is still vulnerable to being stolen and used, all sensitive operations require the `_WL_AUTHCOOKIE_JSESSIONID`, which cannot be stolen, so those operations are protected.

You can also specify a cookie name for `JSESSIONID` or `_WL_AUTHCOOKIE_JSESSIONID` using the `CookieName` parameter defined in the `weblogic.xml` deployment descriptor's <session-descriptor> element, as follows:

```
<session-descriptor>
    <cookie-name>FOOAPPID</cookie-name>
</session-descriptor>
```

In this case, Weblogic Server will not use `JSESSIONID` and `_WL_AUTHCOOKIE_JSESSIONID`, but `FOOAPPID` and `_WL_AUTHCOOKIE_FOOAPPID` to serve the same purpose, as shown in Table 2-1.

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

Table 2-1 WebLogic Server Cookies

User-Specified in Deployment Descriptor	HTTP Session	HTTPS Session
No - uses the JSESSIONID default	JSESSIONID	_WL_AUTHCOOKIE_JSESSIONID
Yes - specified as FOOAPPID	FOOAPPID	_WL_AUTHCOOKIE_FOOAPPID

Developing Secure Web Applications

WebLogic Server supports three types of authentication for Web browsers: BASIC, FORM, and CLIENT-CERT.

The following sections cover the different ways to use these types of authentication:

- Developing BASIC Authentication Web Applications
- Understanding BASIC Authentication with Unsecured Resources
- Developing FORM Authentication Web Applications
- Using Identity Assertion for Web Application Authentication
- Using Two-Way SSL for Web Application Authentication
- Providing a Fallback Mechanism for Authentication Methods
- Developing Swing-Based Authentication Web Applications
- Deploying Web Applications

An alternative way to perform user authentication, including BASIC, FORM, and Custom FORM authentication, is to use the `HttpAuthenticationMechanism` as described in Using Jakarta Security.

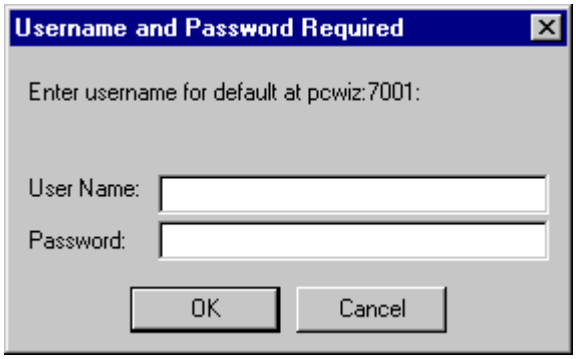
> **Note**
>
> The Jakarta Security specification requires that group principal names are mapped to roles of the same name by default. In WebLogic Server, if the `security-role-assignment` element in the `weblogic.xml` deployment descriptor does not declare a mapping between a security role and one or more principals in the WebLogic Server security realm, then the role name is used as the default principal.

Developing BASIC Authentication Web Applications

With basic authentication, the Web browser pops up a login screen in response to a WebLogic resource request. The login screen prompts the user for a user name and password. Figure 2-2 shows a typical login screen.

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

Figure 2-2 Authentication Login Screen



> **Note**
>
> See Understanding BASIC Authentication with Unsecured Resources for important information about how unsecured resources are handled.

To develop a Web application that provides basic authentication, perform these steps:

1. Create the `web.xml` deployment descriptor. In this file you include the following information (see Example 2-1):
   a. Define the welcome file. The welcome file name is `welcome.jsp`.
   b. Define a security constraint for each set of Web application resources, that is, URL resources, that you plan to protect. Each set of resources share a common URL. URL resources such as HTML pages, JSPs, and servlets are the most commonly protected, but other types of URL resources are supported. In Example 2-1, the URL pattern points to the `welcome.jsp` file located in the Web application's top-level directory; the HTTP methods that are allowed to access the URL resource, POST and GET; and the security role name, `webuser`.

      > **Note**
      >
      > When specifying security role names, observe the following conventions and restrictions:
      >
      > - The proper syntax for a security role name is as defined for an Nmtoken in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
      > - Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
      > - Security role names are case sensitive.
      > - The suggested convention for security role names is that they be singular.

   c. Use the `<login-config>` tag to define the type of authentication you want to use and the security realm to which the security constraints will be applied. Set `<auth-method>BASIC</auth-method>`. The optional `<realm-name>` supplies the realm label

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

shown in the HTTP authentication challenge. WebLogic's active security realm is configured at the domain level and is not selected by this element. See Example 2-1.

d. Define one or more security roles and map them to your security constraints. In our sample, only one security role, `webuser`, is defined in the security constraint so only one security role name is defined here (see the `<security-role>` tag in Example 2-1). However, any number of security roles can be defined.

2. Create the `weblogic.xml` deployment descriptor. In this file you map security role names to users and groups. Example 2-2 shows a sample weblogic.xml file that maps the webuser security role defined in the `<security-role>` tag in the web.xml file to a group named myGroup. Note that principals can be users or groups, so the `<principal-tag>` can be used for either. With this configuration, WebLogic Server will only allow users in myGroup to access the protected URL resource—welcome.jsp.

> **Note**
>
> Starting in version 9.0, the default role mapping behavior is to create empty role mappings when none are specified in weblogic.xml. In version 8.x, if you did not include a weblogic.xml file, or included the file but did not include mappings for all security roles, security roles without mappings defaulted to any user or group whose name matched the role name. For information on role mapping behavior and backward compatibility settings, see Understanding the Combined Role Mapping Enabled Setting in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

3. Create a file that produces the Welcome screen that displays when the user enters a user name and password and is granted access. Example 2-3 shows a sample `welcome.jsp` file.

> **Note**
>
> In Example 2-3, notice that the JSP is calling an API (request.getRemoteUser()) to get the name of the user that logged in. A different API, weblogic.security.Security.getCurrentSubject(), could be used instead. To use this API to get the name of the user, use it with the SubjectUtils API as follows:
>
> ```
> String username = weblogic.security.SubjectUtils.getUsername
> weblogic.security.Security.getCurrentSubject());
> ```

4. Start WebLogic Server and define the users and groups that will have access to the URL resource. In the `weblogic.xml` file (see Example 2-2), the `<principal-name>` tag defines myGroup as the group that has access to the `welcome.jsp`. Therefore, use WebLogic Remote Console to define the `myGroup` group, define a user, and add that user to the `myGroup` group. For information on adding users and groups, see Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

5. Deploy the Web application and use the user defined in the previous step to access the protected URL resource.

   a. For deployment instructions, see Deploying Web Applications.

   b. Open a Web browser and enter this URL:

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

```
http://localhost:7001/basicauth/welcome.jsp
```

c. Enter the user name and password. The Welcome screen displays.

Example 2-1 Basic Authentication web.xml File

```
<?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">

    <welcome-file-list>
        <welcome-file>welcome.jsp</welcome-file>
    </welcome-file-list>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Success</web-resource-name>
            <url-pattern>/welcome.jsp</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>webuser</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>default</realm-name>
    </login-config>

    <security-role>
        <role-name>webuser</role-name>
    </security-role>
</web-app>
```

Example 2-2 BASIC Authentication weblogic.xml File

```
<?xml version='1.0' encoding='UTF-8'?>
<weblogic-web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://
xmlns.oracle.com/weblogic/weblogic-web-app"
         xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-web-app http://
xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">
     <security-role-assignment>
         <role-name>webuser</role-name>
         <principal-name>myGroup</principal-name>
     </security-role-assignment>
</weblogic-web-app>
```

Example 2-3 BASIC Authentication welcome.jsp File

```
<html>
  <head>
    <title>Browser Based Authentication Example Welcome Page</title>
```


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

```
  </head>
  <h1> Browser Based Authentication Example Welcome Page </h1>
  <p> Welcome <%= request.getRemoteUser() %>!
  </blockquote>
  </body>
</html>
```

Using HttpSessionListener to Account for Browser Caching of Credentials

The browser caches user credentials and frequently re-sends them to the server automatically. This can give the appearance that WebLogic Server sessions are not being destroyed after logout or timeout. Depending on the browser, the credentials can be cached just for the current browser session, or across browser sessions.

You can validate that a WebLogic Server's session was destroyed by creating a class that implements the `jakarta.servlet.http.HttpSessionListener` interface. Implementations of this interface are notified of changes to the list of active sessions in a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application in `web.xml`.

To configure a session listener class:

1. Open the `web.xml` deployment descriptor of the Web application for which you are creating a session listener class in a text editor. The `web.xml` file is located in the WEB-INF directory of your Web application.
2. Add an event declaration using the listener element of the web.xml deployment descriptor. The event declaration defines the event listener class that is invoked when the event occurs. For example:

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

   See Configuring an Event Listener Class in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server* for additional information and guidelines.

Write and deploy the session listener class. The example shown in Example 2-4 uses a simple counter to track the session count.

Example 2-4 Tracking the Session Count

```
package myApp;
import jakarta.servlet.http.HttpSessionListener;
import jakarta.servlet.http.HttpSessionEvent;
public class MySessionListener implements HttpSessionListener {
       private static int sessionCount = 0;

       public void sessionCreated(HttpSessionEvent se) {
               sessionCount++;
                // Write to a log or do some other processing.
       }
       public void sessionDestroyed(HttpSessionEvent se) {
                if(sessionCount > 0)
                         sessionCount--;
                     //Write to a log or do some other processing.
           }
}
```


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

Understanding BASIC Authentication with Unsecured Resources

For WebLogic Server versions 9.2 and later, client requests that use HTTP BASIC authentication must pass WebLogic Server authentication, even if access control is not enabled on the target resource.

The setting of the Security Configuration MBean flag enforce-valid-basic-auth-credentials determines this behavior. (The DomainMBean can return the new Security Configuration MBean for the domain.) It specifies whether or not the system should allow requests with invalid HTTP BASIC authentication credentials to access unsecured resources.

> **Note**
>
> The Security Configuration MBean provides domain-wide security configuration information. The enforce-valid-basic-auth-credentials flag effects the entire domain.

The enforce-valid-basic-auth-credentials flag is true by default, and WebLogic Server authentication is performed. If authentication fails, the request is rejected. WebLogic Server must therefore have knowledge of the user and password.

You may want to change the default behavior if you rely on an alternate authentication mechanism. For example, you might use a backend web service to authenticate the client, and WebLogic Server does not need to know about the user. With the default authentication enforcement enabled, the web service can do its own authentication, but only if WebLogic Server authentication first succeeds.

If you explicitly set the enforce-valid-basic-auth-credentials flag to false, WebLogic Server does not perform authentication for HTTP BASIC authentication client requests for which access control was not enabled for the target resource.

In the previous example of a backend web service that authenticates the client, the web service can then perform its own authentication without WebLogic Server having knowledge of the user.

Setting the enforce-valid-basic-auth-credentials Flag

To set the enforce-valid-basic-auth-credentials flag, perform the following steps:

1. Add the `<enforce-valid-basic-auth-credentials>` element to `config.xml` within the `<security-configuration>` element.

   ```
   :
   <enforce-valid-basic-auth-credentials>false</enforce-valid-basic-auth-credentials>
     </security-configuration>
   ```

2. Start or restart all of the servers in the domain.

Check the Value of enforce-valid-basic-auth-credentials

You can use either WebLogic Remote Console or WSLST to check the value of the enforce-valid-basic-auth-credentials setting in a running server. Remember that `enforce-valid-basic-auth-credentials` is a domain-wide setting.

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

In WebLogic Remote Console, go to the Edit Tree perspective, then Environment, then Domain. On the Security tab, click Show Advanced Fields to view the Enforce Valid Basic Auth Credentials option.

The WLST session shown in Example 2-5 demonstrates how to check the value of the enforce-valid-basic-auth-credentials flag in a sample running server.

Example 2-5 Checking the Value of enforce-valid-basic-auth-credentials

```
wls:/offline> connect('','','t3://host:port')
Please enter your username :adminuser
Please enter your password :
Connecting to t3://host:port with userid adminuser ...
Successfully connected to Admin Server 'examplesServer' that belongs to domain '
wl_server'.
wls:/wl_server/serverConfig> cd('SecurityConfiguration')

wls:/wl_server/serverConfig/SecurityConfiguration> ls()
dr--   wl_server
wls:/wl_server/serverConfig/SecurityConfiguration> cd('wl_server')
wls:/wl_server/serverConfig/SecurityConfiguration/wl_server> ls()
dr--   DefaultRealm
dr--   Realms
-r--   AnonymousAdminLookupEnabled                 false
-r--   CompatibilityConnectionFiltersEnabled       false
-r--   ConnectionFilter                            null
-r--   ConnectionFilterRules                       null
-r--   ConnectionLoggerEnabled                     false
-r--   ConsoleFullDelegationEnabled                false
-r--   Credential                                  ******
-r--   CredentialEncrypted                         ******
-r--   CrossDomainSecurityEnabled                  false
-r--   DowngradeUntrustedPrincipals                false
-r--   EnforceStrictURLPattern                     true
-r--   EnforceValidBasicAuthCredentials            false
:
:
```

Developing FORM Authentication Web Applications

When using FORM authentication with Web applications, you provide a custom login screen that the Web browser displays in response to a Web application resource request and an error screen that displays if the login fails. The login screen can be generated using an HTML page, JSP, or servlet. The benefit of form-based login is that you have complete control over these screens so that you can design them to meet the requirements of your application or enterprise policy/guideline.

The login screen prompts the user for a user name and password. Figure 2-3 shows a typical login screen generated using a JSP and Example 2-6 shows the source code.

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

Figure 2-3 Form-Based Login Screen (login.jsp)



Figure 2-4 shows a typical login error screen generated using HTML and Example 2-7 shows the source code.

Figure 2-4 Login Error Screen



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

To develop a Web application that provides FORM authentication, perform these steps:

1. Create the `web.xml` deployment descriptor and include the following information:
   a. Define the welcome file. The welcome file name is `welcome.jsp`.
   b. Define a security constraint for each set of URL resources that you plan to protect. Each set of URL resources share a common URL. URL resources such as HTML pages, JSPs, and servlets are the most commonly protected, but other types of URL resources are supported. In the sample `web.xml` file provided in the following steps, the URL pattern points to /admin/edit.jsp, thus protecting the `edit.jsp` file located in the Web application's `admin` sub-directory, defines the HTTP method that is allowed to access the URL resource, `GET`, and defines the security role name, `admin`.

      Note

      Do not use hyphens in security role names. Security role names with hyphens cannot be modified in WebLogic Remote Console. Also, the suggested convention for security role names is that they be singular.

   c. Define the type of authentication you want to use and the security realm to which the security constraints will be applied. In this case, the `FORM` type is specified and no realm is specified, so the realm is the default realm, which means that the security constraints will apply to the security realm that is activated when a WebLogic Server instance boots.
   d. Define one or more security roles and map them to your security constraints. In our sample, only one security role, `admin`, is defined in the security constraint so only one security role name is defined here. However, any number of security roles can be defined. The following is a sample `web.xml` file.

```
<?xml version='1.0' encoding='UTF-8'?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/j2ee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <welcome-file-list>
        <welcome-file>welcome.jsp</welcome-file>
    </welcome-file-list>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>AdminPages</web-resource-name>
            <description>
                These pages are only accessible by authorized
                 administrators.
            </description>
            <url-pattern>/admin/edit.jsp</url-pattern>
         </web-resource-collection>
        <auth-constraint>
            <description>
                These are the roles who have access.
            </description>
            <role-name>
                admin
            </role-name>
        </auth-constraint>
        <user-data-constraint>
            <description>
```


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

```
                    This is how the user data must be transmitted.
                </description>
                <transport-guarantee>NONE</transport-guarantee>
            </user-data-constraint>
        </security-constraint>
        <login-config>
            <auth-method>FORM</auth-method>
            <form-login-config>
                <form-login-page>/login.jsp</form-login-page>
                <form-error-page>/fail_login.html</form-error-page>
            </form-login-config>
        </login-config>
        <security-role>
            <description>
                An administrator
            </description>
            <role-name>
                admin
            </role-name>
        </security-role>
    </web-app>
```

2. Create the `weblogic.xml` deployment descriptor as shown in the following example. In this file, you map security role names to users and groups. The following example shows a sample `weblogic.xml` file that maps the `admin` security role defined in the `<security-role>` tag in the `web.xml` file to the group supportGroup. With this configuration, WebLogic Server will only allow users in the supportGroup group to access the protected WebLogic resource.

```
<?xml version='1.0' encoding='UTF-8'?>
<weblogic-web-app xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <security-role-assignment>
         <role-name>admin</role-name>
         <principal-name>supportGroup</principal-name>
     </security-role-assignment>
</weblogic-web-app>
```

   However, you can use WebLogic Remote Console to modify the Web application's security role so that other groups can be allowed to access the protected WebLogic resource.

3. Create a Web application file that produces the welcome screen when the user requests the protected Web application resource by entering the URL. The following example shows a sample `welcome.jsp` file.

```
<html>
  <head>
    <title>Security login example</title>
  </head>
  <%
    String bgcolor;
    if ((bgcolor=(String)application.getAttribute("Background")) ==
        null)
    {
        bgcolor="#cccccc";
    }
```


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

```
  %>
  <body bgcolor=<%="\""+bgcolor+"\""%>>
  <blockquote>
  <img src=Button_Final_web.gif align=right>
  <h1> Security Login Example </h1>
  <p> Welcome <%= request.getRemoteUser() %>!
  <p> If you are an administrator, you can configure the background
  color of the Web Application.
  <br> <b><a href="admin/edit.jsp">Configure background</a></b>.
  <% if (request.getRemoteUser() != null) { %>
    <p> Click here to <a href="logout.jsp">logout</a>.
  <% } %>
  </blockquote>
  </body>
</html>
```

> **Note**
>
> In Example 2-3, notice that the JSP is calling an API (request.getRemoteUser()) to get the name of the user that logged in. A different API, weblogic.security.Security.getCurrentSubject(), could be used instead. To use this API to get the name of the user, use it with the SubjectUtils API as follows:
>
> ```
> String username = weblogic.security.SubjectUtils.getUsername(
> weblogic.security.Security.getCurrentSubject());
> ```

4. Start WebLogic Server and define the users and groups that will have access to the URL resource. In the sample `weblogic.xml` file, the `<role-name>` tag defines admin as the group that has access to the `edit.jsp` file and defines the user, 'joe' as a member of that group. Therefore, use WebLogic Remote Console to define the admin group, and define the user 'joe' and add 'joe' to the admin group. You can also define other users and add them to the group to grant them access to the protected WebLogic resource. For information on adding users and groups, see Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.
5. Deploy the Web application and use the user defined in the previous step to access the protected Web application resource.
   a. For deployment instructions, see Deploying Web Applications.
   b. Open a Web browser and enter this URL:

      ```
      http://hostname:7001/security/welcome.jsp
      ```

   c. Enter the user name and password. The Welcome screen displays.

Example 2-6 Form-Based Login Screen Source Code (login.jsp)

```
<html>
  <head>)
    <title>Security WebApp login page</title>
  </head>
  <body bgcolor="#cccccc">
  <blockquote>
  <img src=Button_Final_web.gif align=right>
  <h2>Please enter your user name and password:</h2>
  <p>
```


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

```
  <form method="POST" action="j_security_check">
  <table border=1>
    <tr>
      <td>Username:</td>
      <td><input type="text" name="j_username"></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type="password" name="j_password"></td>
    </tr>
    <tr>
      <td colspan=2 align=right><input type=submit
                                       value="Submit"></td>
    </tr>
  </table>
  </form>
  </blockquote>
  </body>
</html>
```

Example 2-7 Login Error Screen Source Code

```
<html>
  <head>
    <title>Login failed</title>
  </head>
  <body bgcolor=#ffffff>
  <blockquote>
  <img src=/security/Button_Final_web.gif align=right>
  <h2>Sorry, your user name and password were not recognized.</h2>
  <p><b>
  <a href="/security/welcome.jsp">Return to welcome page</a> or
           <a href="/security/logout.jsp">logout</a>
  </b>
  </blockquote>
  </body>
</html>
```

Using Identity Assertion for Web Application Authentication

You use identity assertion in Web applications to verify client identities for authentication purposes. When using identity assertion, the following requirements must be met:

1. The authentication type must be set to CLIENT-CERT.
2. An Identity Assertion provider must be configured in the server. If the Web browser or Java client requests a WebLogic Server resource protected by a security policy, WebLogic Server requires that the Web browser or Java client have an identity. The WebLogic Identity Assertion provider maps the token from a Web browser or Java client to a user in a WebLogic Server security realm. For information on how to configure an Identity Assertion provider, see Configuring Identity Assertion Providers in *Administering Security for Oracle WebLogic Server*.
3. The user corresponding to the token's value must be defined in the server's security realm; otherwise the client will not be allowed to access a protected WebLogic resource. For information on configuring users on the server, see Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

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

Using Two-Way SSL for Web Application Authentication

You use two-way SSL in Web applications to verify that clients are whom they claim to be. When using two-way SSL, the following requirements must be met:

1. The authentication type must be set to CLIENT-CERT.
2. The server must be configured for two-way SSL. For information on using SSL and digital certificates, see Using SSL Authentication in Java Clients. For information on configuring SSL on the server, see Configuring SSL in *Administering Security for Oracle WebLogic Server*.
3. The client must use HTTPS to access the Web application on the server.
4. An Identity Assertion provider must be configured in the server. If the Web browser or Java client requests a WebLogic Server resource protected by a security policy, WebLogic Server requires that the Web browser or Java client have an identity. The WebLogic Identity Assertion provider allows you to enable a user name mapper in the server that maps the digital certificate of a Web browser or Java client to a user in a WebLogic Server security realm. For information on how to configure security providers, see Configuring WebLogic Security Providers in *Administering Security for Oracle WebLogic Server*.
5. The user corresponding to the Subject's Distinguished Name (SubjectDN) attribute in the client's digital certificate must be defined in the server's security realm; otherwise the client will not be allowed to access a protected WebLogic resource. For information on configuring users on the server, see Users, Groups, and Security Roles in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

> **Note**
>
> When you use SSL authentication, it is not necessary to use web.xml and weblogic.xml files to specify server configuration because you use WebLogic Remote Console to specify the server's SSL configuration.

Providing a Fallback Mechanism for Authentication Methods

The Jakarta Servlet 5.0 specification (https://jakarta.ee/specifications/servlet/) allows you to define the authentication method (BASIC, FORM, etc.) to be used in a Web application. WebLogic Server provides an `auth-method` security module that allows you to define multiple authentication methods (as a comma separated list), so the container can provide a fall-back mechanism. Authentication will be attempted in the order the values are defined in the `auth-method` list.

For example, you can define the following `auth-method` list in the `login-config` element of your `web.xml` file:

```
<login-config>
  <auth-method>CLIENT-CERT,BASIC</auth-method>
</login-config>
```

Then the container will first try to authenticate by looking at the CLIENT-CERT value. If that should fail, the container will challenge the user-agent for BASIC authentication.

If either FORM or BASIC are configured, then they must exist at the end of the list since they require a round-trip communication with the user. However, both FORM and BASIC cannot exist together in the list of `auth-method` values.

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

Configuration

The `auth-method` authentication security can be configured in two ways:

- Define a comma separated list of `auth-method` values in the `login-config` element of your `web.xml` file.
- Define the `auth-method` values as a comma separated list on the `RealmMBean` and in the `login-config` element of your `web.xml` use the REALM value, then the Web application will pick up the authentication methods from the security realm.

WebLogic Java Management Extensions (JMX) enables you to access the RealmMBean to create and manage the security resources. For more information, see Overview of WebLogic Server Subsystem MBeans in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

Developing Swing-Based Authentication Web Applications

Web browsers can also be used to run graphical user interfaces (GUIs) that were developed using Java Foundation Classes (JFC) Swing components.

For information on how to create a graphical user interface (GUI) for applications and applets using the Swing components, see the *Creating a GUI with JFC/Swing* tutorial (also known as The Swing Tutorial). You can access this tutorial on the Web at `http://docs.oracle.com/javase/tutorial/uiswing/`.

After you have developed your Swing-based GUI, refer to Developing FORM Authentication Web Applications and use the Swing-based screens to perform the steps required to develop a Web application that provides FORM authentication.

> **Note**
>
> When developing a Swing-based GUI, do not rely on the Java Virtual Machine-wide user for child threads of the swing event thread. This is not Jakarta EE compliant and does not work in thin clients, or in IIOP in general. Instead, take either of the following approaches:
>
> - Make sure an InitialContext is created before any Swing artifacts.
> - Or, use the Java Authentication and Authorization Service (JAAS) to log in and then use the Security.runAs() method inside the Swing event thread and its children.

Deploying Web Applications

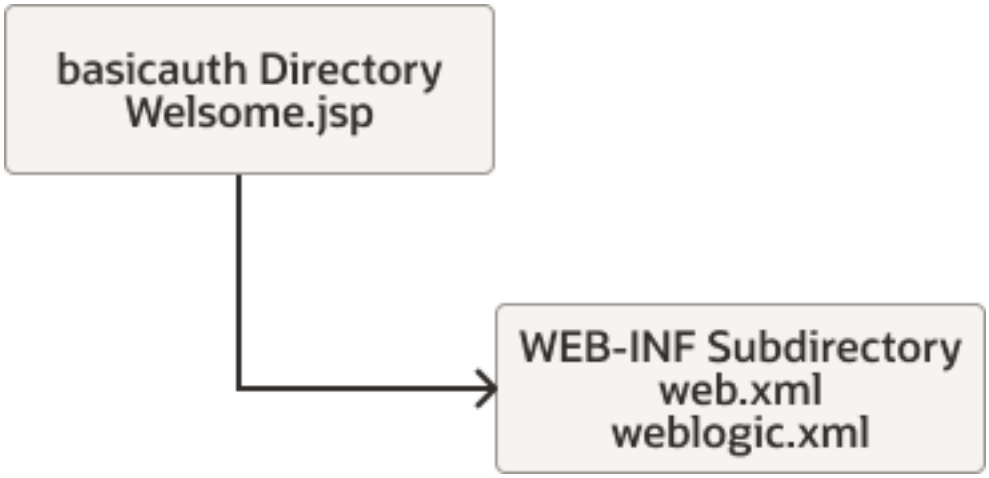
To deploy a Web application on a server running in development mode, perform the following steps:

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

> **Note**
>
> For more information about deploying Web applications in either development or production mode, see Deploying Applications and Modules with weblogic.deployer in *Deploying Applications to Oracle WebLogic Server*.

1. Set up a directory structure for the Web application's files. Figure 2-5 shows the directory structure for the Web application named `basicauth`. The top-level directory must be assigned the name of the Web application and the sub-directory must be named `WEB-INF`.

Figure 2-5 Basicauth Web Application Directory Structure



2. To deploy the Web application in exploded directory format, that is, not in the Java archive (jar) format, simply move your directory to the `applications` directory on your server. For example, you would deploy the `basicauth` Web application in the following location:

   ```
   ORACLE_HOME\user_projects\domains\mydomain\applications\basicauth
   ```

   If the WebLogic Server instance is running, the application should auto-deploy. Use WebLogic Remote Console to verify that the application deployed.

   If the WebLogic Server instance is not running, the Web application should auto-deploy when you start the server.

3. If you have not done so already, use WebLogic Remote Console to configure the users and groups that will have access to the Web application. To determine the users and groups that are allowed access to the protected WebLogic resource, examine the `weblogic.xml` file. For example, the `weblogic.xml` file for the `basicauth` sample (see Example 2-2) defines `myGroup` as the only group to have access to the `welcome.jsp` file.

For more information on deploying secure Web applications, see Deploying Applications and Modules with weblogic.deployer in *Deploying Applications to Oracle WebLogic Server*.

Using Declarative Security With Web Applications

WebLogic Server supports three different ways to implement declarative security web applications. You can define policies and roles using WebLogic Remote Console, you can use

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

Jakarta Authorization to configure a Java permission-based security model, or you can configure security entirely within the web application's deployment descriptor files.

For information about using Jakarta Authorization, see Using Jakarta Authorization. The topics that follow explain how to configure security in web application's deployment descriptors.

Which of these three methods is used is defined by the Jakarta Authorization flags and the security model. (Security models are described in Options for Securing EJB and Web Application Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.)

To implement declarative security in Web applications, you can use deployment descriptors (`web.xml` and `weblogic.xml`) to define security requirements. The deployment descriptors map the application's logical security requirements to its runtime definitions. And at runtime, the servlet container uses the security definitions to enforce the requirements. For a discussion of using deployment descriptors, see Developing Secure Web Applications.

For information about how to use deployment descriptors and the `externally-defined` element to configure security in Web applications declaratively, see externally-defined.

WebLogic Server supports several deployment descriptor elements that are used in the `web.xml` and `weblogic.xml` files to define security requirements in Web applications.

Web Application Security-Related Deployment Descriptors

WebLogic Server supports several deployment descriptor elements that are used in the `web.xml` and `weblogic.xm` files to define security requirements in Web applications.

- web.xml Deployment Descriptors
- weblogic.xml Deployment Descriptors

web.xml Deployment Descriptors

The following `web.xml` security-related deployment descriptor elements are supported by WebLogic Server:

- auth-constraint
- security-constraint
- security-role
- security-role-ref
- user-data-constraint
- web-resource-collection

auth-constraint

The optional `auth-constraint` element defines which groups or principals have access to the collection of Web resources defined in this security constraint.

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

> **Note**
>
> Any resource that is protected by an `auth-constraint` element should also be protected by a Table 2-6 with a `<transport-guarantee>` of `INTEGRAL` or `CONFIDENTIAL`.
>
> WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the `INTEGRAL` or `CONFIDENTIAL` transport guarantee, thereby ensuring that all communication on the network between the Web browser and the server is encrypted and that none of the communication, including a user name and password, is in clear text.
>
> Requiring SSL also means that WebLogic Server uses two cookies: the `JSESSIONID` cookie and the encrypted `_WL_AUTHCOOKIE_JSESSIONID` cookie, as described in Using Secure Cookies to Prevent Session Stealing.

The following table describes the elements you can define within an `auth-constraint` element.

Table 2-2 auth-constraint Element

Element	Required/Optional	Description
`<description>`	Optional	A text description of this security constraint.
`<role-name>`	Optional	Defines which security roles can access resources defined in this `<security-constraint>`. Security role names are mapped to principals using the `<security-role-ref>` element. See security-role-ref.

Used Within

The `auth-constraint` element is used within the `security-constraint` element.

Example

See Example 2-8 for an example of how to use the `auth-constraint` element in a `web.xml` file.

security-constraint

The `security-constraint` element is used in the `web.xml` file to define the access privileges to a collection of resources defined by the `web-resource-collection` element.

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

> **Note**
>
> Any resource that is protected by an `auth-constraint` element should also be protected by a Table 2-6 with a `<transport-guarantee>` of `INTEGRAL` or `CONFIDENTIAL`.
>
> WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the `INTEGRAL` or `CONFIDENTIAL` transport guarantee, thereby ensuring that all communication on the network between the Web browser and the server is encrypted and that none of the communication, including a user name and password, is in clear text.
>
> Requiring SSL also means that WebLogic Server uses two cookies: the `JSESSIONID` cookie and the encrypted `_WL_AUTHCOOKIE_JSESSIONID` cookie, as described in Using Secure Cookies to Prevent Session Stealing.

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

Table 2-3 security-constraint Element

Element	Required/Optional	Description
`<web-resource-collection>`	Required	Defines the components of the Web Application to which this security constraint is applied. See web-resource-collection.
`<auth-constraint>`	Optional	Defines which groups or principals have access to the collection of web resources defined in this security constraint. See auth-constraint.
`<user-data-constraint>`	Optional	Defines how data communicated between the client and the server should be protected. See user-data-constraint.

Example

Example 2-8 shows how to use the `security-constraint` element to defined security for the SecureOrdersEast resource in a `web.xml` file.

Example 2-8 Security Constraint Example

```
web.xml entries:
<security-constraint>
          <web-resource-collection>
          <web-resource-name>SecureOrdersEast</web-resource-name>
          <description>
             Security constraint for
             resources in the orders/east directory
          </description>
          <url-pattern>/orders/east/*</url-pattern>
          <http-method>POST</http-method>
          <http-method>GET</http-method>
     </web-resource-collection>
     <auth-constraint>
          <description>
           constraint for east coast sales
          </description>
          <role-name>east</role-name>
          <role-name>manager</role-name>
     </auth-constraint>
```


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

```
 <user-data-constraint>
          <description>SSL not required</description>
          <transport-guarantee>NONE</transport-guarantee>
     </user-data-constraint>
</security-constraint>
...
```

security-role

The `security-role` element contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name.

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

Table 2-4 security-role Element

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

Example

See Example 2-11 for an example of how to use the `security-role` element in a `web.xml` file.

security-role-ref

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

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

Table 2-5 security-role-ref Element

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

Example

See isUserInRole for an example of how to use the `security-role-ref` element in a `web.xml` file.

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

user-data-constraint

The `user-data-constraint` element defines how data communicated between the client and the server should be protected.

> **Note**
>
> Any resource that is protected by an `auth-constraint` element should also be protected by a Table 2-6 with a `<transport-guarantee>` of `INTEGRAL` or `CONFIDENTIAL`.
>
> WebLogic Server establishes a Secure Sockets Layer (SSL) connection when the user is authenticated using the `INTEGRAL` or `CONFIDENTIAL` transport guarantee, thereby ensuring that all communication on the network between the Web browser and the server is encrypted and that none of the communication, including a user name and password, is in clear text.
>
> Requiring SSL also means that WebLogic Server uses two cookies: the `JSESSIONID` cookie and the encrypted `_WL_AUTHCOOKIE_JSESSIONID` cookie, as described in Using Secure Cookies to Prevent Session Stealing.

The following table describes the elements you can define within a `user-data-constraint` element.

Table 2-6 user-data-constraint Element

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

Used Within

The `user-data-constraint` element is used within the `security-constraint` element.

Example

See Example 2-8 for an example of how to use the `user-data-constraint` element in a `web.xml` file.

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

web-resource-collection

The `web-resource-collection` element identifies a subset of the resources and `HTTP` methods on those resources within a Web application to which a security constraint applies. If no `HTTP` methods are specified, the security constraint applies to all `HTTP` methods.

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

Table 2-7 web-resource-collection Element

Element	Required/Optional	Description
`<web-resource-name>`	Required	The name of this web resource collection.
`<description>`	Optional	Text description of the Web resource.
`<url-pattern>`	Required	The mapping, or location, of the Web resource collection. URL patterns must use the syntax defined in the Java Servlet Specification (https://jakarta.ee/specifications/servlet/). The pattern `<url-pattern>/</url-pattern>` applies the security constraint to the entire Web application.
`<http-method>`	Optional	The `HTTP` methods to which the security constraint applies when clients attempt to access the Web resource collection. If no `HTTP` methods are specified, then the security constraint applies to all `HTTP` methods. Specifying an `HTTP` method here limits the reach of the security constraint. Unless you have a particular requirement to specify an `HTTP` method, for security reasons you should not set this element.

Used Within

The `web-resource-collection` element is used within the `security-constraint` element.

Example

See Example 2-8 for an example of how to use the `web-resource-collection` element in a `web.xml` file.

weblogic.xml Deployment Descriptors

The following `weblogic.xml` security-related deployment descriptor elements are supported by WebLogic Server:

- externally-defined
- run-as-principal-name
- run-as-role-assignment
- security-permission

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

- security-permission-spec
- security-role-assignment

For additional information on `weblogic.xml` deployment descriptors, see XML Deployment Descriptors in *Developing Applications for Oracle WebLogic Server*.

For additional information on the `weblogic.xml` elements, see weblogic.xml Deployment Descriptor Elements in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

externally-defined

The `externally-defined` element lets you explicitly indicate that you want the security roles defined by the `role-name` element in the `web.xml` deployment descriptors to use the mappings specified in WebLogic Remote Console. The element gives you the flexibility of not having to specify a specific security role mapping for each security role defined in the deployment descriptors for a particular Web application. Therefore, within the same security realm, deployment descriptors can be used to specify and modify security for some applications while WebLogic Remote Console can be used to specify and modify security for others.

The role mapping behavior for a server depends on which security deployment model is selected in WebLogic Remote Console. For information on security deployment models, see Options for Securing EJB and Web Application Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

> **Note**
>
> When specifying security role names, observe the following conventions and restrictions:
>
> - The proper syntax for a security role name is as defined for an Nmtoken in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
> - Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
> - Security role names are case sensitive.
> - The suggested convention for security role names is that they be singular.

Used Within

The `externally-defined` element is used within the `security-role-assignment` element.

Example

Example 2-9 and Example 2-10 show by comparison how to use the `externally-defined element` in the `weblogic.xml` file. In Example 2-10, the specification of the "webuser" `externally-defined` element in the `weblogic.xml` means that for security to be correctly configured on the `getReceipts` method, the principals for `webuser` will have to be created in WebLogic Remote Console.

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

> **Note**
>
> If you need to list a significant number of principals, consider specifying groups instead of users. There are performance issues if you specify too many users.

Example 2-9 Using the web.xml and weblogic.xml Files to Map Security Roles and Principals to a Security Realm

```
web.xml entries:
<web-app>
           ...
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
           ...
</web-app>
<weblogic.xml entries:
<weblogic-web-app>
     <security-role-assignment>
         <role-name>webuser</role-name>
         <principal-name>myGroup</principal-name>
         <principal-name>Bill</principal-name>
         <principal-name>Mary</principal-name>
     </security-role-assignment>
</weblogic-web-app>
```

Example 2-10 Using the externally-defined tag in Web Application Deployment Descriptors

```
web.xml entries:
<web-app>
           ...
           <security-role>
               <role-name>webuser</role-name>
           </security-role>
           ...
</web-app>
<weblogic.xml entries:
<weblogic-web-app>
     <security-role-assignment>
         <role-name>webuser</role-name>
         <externally-defined/>
     </security-role-assignment>
```

For information about how to use WebLogic Remote Console to configure security for Web applications, see Securing Web Applications and EJBs in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

run-as-principal-name

The `run-as-principal-name` element specifies the name of a principal to use for a security role defined by a `run-as` element in the companion `web.xml` file.

Used Within

The `run-as-principal-name` element is used within a `run-as-role-assignment` element.

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

Example

For an example of how to use the `run-as-principal-name` element, see Example 2-11.

run-as-role-assignment

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

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

Table 2-8 run-as-role-assignment Element

Element	Required/Optional	Description
`<role-name>`	Required	Specifies the name of a security role name specified in a `run-as` element in the companion `web.xml` file.
`<run-as-principal-name>`	Required	Specifies a principal for the security role name defined in a `run-as` element in the companion `web.xml` file.

Example:

Example 2-11 shows how to use the `run-as-role-assignment` element to have the `SnoopServlet` always execute as a user `joe`.

Example 2-11 run-as-role-assignment Element Example

```
web.xml:
  <servlet>
    <servlet-name>SnoopServlet</servlet-name>
    <servlet-class>extra.SnoopServlet</servlet-class>
    <run-as>
      <role-name>runasrole</role-name>
    </run-as>
  </servlet>
  <security-role>
    <role-name>runasrole</role-name>
  </security-role>
weblogic.xml:
  <weblogic-web-app>
     <run-as-role-assignment>
       <role-name>runasrole</role-name>
       <run-as-principal-name>joe</run-as-principal-name>
     </run-as-role-assignment>
  </weblogic-web-app>
```

security-permission

The `security-permission` element specifies a security permission that is associated with a Jakarta EE Sandbox.

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

Example

For an example of how to used the `security-permission` element, see Example 2-12.

security-permission-spec

The security-permission-spec element specifies a single security permission based on the Security policy file syntax. Refer to the *Default Policy Implementation and Policy File Syntax* section in Java SE Security Developer's Guide for the implementation of the security permission specification.

> **Note**
>
> Disregard the optional codebase and signedBy clauses.

Used Within

The security-permission-`spec` element is used within the `security-permission` element.

Example

Example 2-12 shows how to use the security-permission-spec element to grant permission to the `java.net.SocketPermission` class.

Example 2-12 security-permission-spec Element Example

```
<weblogic-web-app>
   <security-permission>
     <description>Optional explanation goes here</description>
     <security-permission-spec>
<!--
A single grant statement following the syntax of
http://xmlns.jcp.org/j2se/1.5.0/docs/guide/security/PolicyFiles.html#FileSyntax,
without the "codebase" and "signedBy" clauses, goes here. For example:
-->
      grant {
      permission java.net.SocketPermission "*", "resolve";
      };
     </security-permission-spec>
   </security-permission>
</weblogic-web-app>
```

In Example 2-12, permission `java.net.SocketPermission` is the permission class name, "*" represents the target name, and resolve indicates the action (resolve host/IP name service lookups).

security-role-assignment

The `security-role-assignment` element declares a mapping between a security role and one or more principals in the WebLogic Server security realm.

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

> **Note**
>
> For information on using the security-role-assignment element in a weblogic-application.xml deployment descriptor for an enterprise application, see Enterprise Application Deployment Descriptor Elements in *Developing Applications for Oracle WebLogic Server*.

Example

Example 2-13 shows how to use the `security-role-assignment` element to assign principals to the `PayrollAdmin` role.

> **Note**
>
> If you need to list a significant number of principals, consider specifying groups instead of users. There are performance issues if you specify too many users.

Example 2-13 security-role-assignment Element Example

```
<weblogic-web-app>
  <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>
</weblogic-web-app>
```

Using Programmatic Security With Web Applications

You can write your servlets to access users and security roles programmatically using methods defined in the Jakarta Security specification `SecurityContext` interface and the Servlet `HttpServletRequest` interface.

These sections describe the methods in more detail:

- Jakarta Security SecurityContext Methods
- Servlet HttpServletRequest Methods

Jakarta Security SecurityContext Methods

WebLogic Server supports these Jakarta Security SecurityContext methods in the Servlet (including Webservice) and EJB containers, as specified in the Java specification:

- `getCallerPrincipal()` - Use this method to retrieve the Principal representing the caller. This is the container-specific representation of the caller principal. The type may differ from the type of the caller principal originally established by an `HttpAuthenticationMechanism`. This method returns null for an unauthenticated caller in either the Servlet Container or the EJB Container.
- `getPrincipalsByType()` - Use this method to retrieve all principals of the given type. It can be used to retrieve an application-specific caller principal established during authentication.

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

This method is primarily useful when the container's caller principal is a different type than the application caller principal, and the application needs specific information behavior available only from the application principal. This method returns an empty Set if the caller is unauthenticated, or if the requested type is not found.

- `isCallerInRole()` - Use this method to check if the authenticated caller is included in the specified logical application "role". The method takes a String argument that represents the specific role to be verified.
- `hasAccessToWebResource()` - Use this method to determine if the caller has access to the specified web resource for the specified HTTP methods, as determined by the security constraints configured for the application. The resource parameter is a `URLPatternSpec`, as defined by the Jakarta Authorization specification (https://jakarta.ee/specifications/authorization), that identifies an application-specific web resource. This method can be used to check access to resources in the current application only — it cannot be called cross-application, or cross-container, to check access to resources in a different application.
- `authenticate()` - Use this method to signal to the container that it should start the authentication process with the caller.

Servlet HttpServletRequest Methods

You can write your servlets to access users and security roles programmatically in your servlet code by using the `jakarta.servlet.http.HttpServletRequest.getUserPrincipal` and `jakarta.servlet.http.HttpServletRequest.isUserInRole(String role)` methods.

- getUserPrincipal
- isUserInRole

getUserPrincipal

You use the `getUserPrincipal()` method to determine the current user of the Web application. This method returns a `WLSUser Principal` if one exists in the current user. In the case of multiple `WLSUser Principals`, the method returns the first in the ordering defined by the `Subject.getPrincipals().iterator()` method. If there are no `WLSUser Principals`, then the `getUserPrincipal()` method returns the first non-`WLSGroup Principal`. If there are no `Principals` or all `Principals` are of type `WLSGroup`, this method returns `null`. This behavior is identical to the semantics of the `weblogic.security.SubjectUtils.getUserPrincipal()` method.

For more information about how to use the `getUserPrincipal()` method, see Jakarta EE Platform API.

isUserInRole

`The jakarta.servlet.http.HttpServletRequest.isUserInRole(String role)` method returns a boolean indicating whether the authenticated user is granted the specified logical security "role." If the user has not been authenticated, this method returns false.

The `isUserInRole()` method maps security roles to the group names in the security realm. The following example shows the elements that are used with the `<servlet>` element to define the security role in the `web.xml` file.

```
Begin web.xml entries:
...
```


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

```
<servlet>
        <security-role-ref>
             <role-name>user-rolename</role-name>
             <role-link>rolename-link</role-link>
         </security-role-ref>
</servlet>
<security-role>
         <role-name>rolename-link</role-name>
</security-role>
...
Begin weblogic.xml entries:
...
<security-role-assignment>
         <role-name>rolename-link</role-name>
         <principal-name>groupname</principal>
         <principal-name>username</principal>
</security-role-assignment>
...
```

In this example, the string `role` is mapped to the name supplied in the `<role-name>` element, which is nested inside the `<security-role-ref>` element of a `<servlet>` declaration in the `web.xml` deployment descriptor. The `<role-name>` element defines the name of the security role or `principal` (the user or group) that is used in the servlet code. The `<role-link>` element maps to a `<role-name>` defined in the `<security-role-assignment>` element in the `weblogic.xml` deployment descriptor.

> **Note**
>
> When specifying security role names, observe the following conventions and restrictions:
>
> - The proper syntax for a security role name is as defined for an Nmtoken in the Extensible Markup Language (XML) recommendation available on the Web at: http://www.w3.org/TR/REC-xml#NT-Nmtoken.
> - Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
> - Security role names are case sensitive.
> - The suggested convention for security role names is that they be singular.

For example, if the client has successfully logged in as user `Bill` with the security role of `manager`, the following method would return true:

```
request.isUserInRole("manager")
```

Example 2-14 provides an example.

Example 2-14 Example of Security Role Mapping

```
Servlet code:
out.println("Is the user a Manager? " +
             request.isUserInRole("manager"));
web.xml entries:
<servlet>
. . .
```


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

```
   <role-name>manager</role-name>
   <role-link>mgr</role-link>
. . .
</servlet>
<security-role>
   <role-name>mgr</role-name>
</security-role>
weblogic.xml entries:
<security-role-assignment>
   <role-name>mgr</role-name>
   <principal-name>bostonManagers</principal-name>
   <principal-name>Bill</principal-name>
   <principal-name>Ralph</principal-name>
</security-role-ref>
```

Authenticating Users Programmatically

WebLogic Server supports programmatic authentication within a servlet application using the Jakarta Security `SecurityContext` interface or the WebLogic Server `ServletAuthentication` API.

Topics

- Using the Jakarta Security SecurityContext Interface
- Using the Programmatic Authentication API

Using the Jakarta Security SecurityContext Interface

WebLogic Server supports the `authenticate` method of the Jakarta Security `SecurityContext` interface for authenticating users. The `authenticate` method is enabled by default in the Servlet container, and is useful when you need a portable authentication solution.

You use the `authenticate()` method in the application to signal to the container that it should start the authentication process with the caller. This method can only be used in a valid `ServletContext` because it requires that `HttpServletRequest` and `HttpServletResponse` parameters are passed in.

Using the Programmatic Authentication API

WebLogic Server provides a server-side `weblogic.servlet.security.ServletAuthentication` API that supports programmatic authentication from within a servlet application.

You can use the `weblogic.servlet.security.ServletAuthentication` API to authenticate and log in the user. Once the login is completed, it appears as if the user logged in using the standard mechanism.

You have the option of using either of two WebLogic-supplied classes with the `ServletAuthentication` API, the `weblogic.security.SimpleCallbackHandler` class or the `weblogic.security.URLCallbackHandler` class. For more information on these classes, see *Java API Reference for Oracle WebLogic Server*.

Example 2-15 shows an example that uses `SimpleCallbackHandler`. Example 2-16 shows an example that uses `URLCallbackHandler`.

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

Example 2-15 Programmatic Authentication Code Fragment Using the SimpleCallbackHandler Class

```
CallbackHandler handler = new SimpleCallbackHandler(username,
                                                    password);
Subject mySubject =
        weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
// Where request is the httpservletrequest object.
```

Example 2-16 Programmatic Authentication Code Fragment Using the URLCallbackHandler Class

```
CallbackHandler handler = new URLCallbackHandler(username,
                                                 password);
Subject mySubject =
        weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
// Where request is the httpservletrequest object.
```

Change the User's Session ID at Login

When an HttpSession is created in a servlet, it is associated with a unique ID. The browser must provide this session ID with its request in order for the server to find the session data again.

In order to avoid a type of attack called "session fixation," you should change the user's session ID at login. To do this, call the `generateNewSessionID` method of `weblogic.servlet.security.ServletAuthentication` after you call the `login` method.

The `generateNewSessionID` method moves all current session information into a completely different session ID and associates this session with this new ID.

> **Note**
>
> The session itself does not change, only its identifier.

It is possible that legacy applications might depend on the session ID remaining the same before and after login. Calling `generateNewSessionID` would break such an application. Oracle recommends that you do not build this dependency into your application. However, if you do, or if you are dealing with a legacy application of this type, Oracle recommends that you use SSL to protect all access to the application.

Note that, by default, the WebLogic container automatically regenerates IDs for non-programmatic logins.

See `ServletAuthentication` for additional information about the `generateNewSessionID()` method.

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

3

Using JAAS Authentication in Java Clients

Oracle WebLogic Server provides support for using JAAS authentication in Java clients. Learn how to implement this type of authentication.

- JAAS and WebLogic Server
- JAAS Authentication Development Environment
- Writing a Client Application Using JAAS Authentication
- Using JNDI Authentication
- Java Client JAAS Authentication Code Examples

The sections refer to sample code which is included in the WebLogic Server distribution at:

```
EXAMPLES_HOME\src\examples\security\jaas
```

The *EXAMPLES_HOME* directory can be found at *ORACLE_HOME*`\wlserver\samples\server`.

The `jaas` directory contains an `instructions.html` file, `ant` build files, a `sample_jaas.config` file, and the following Java files:

- `BaseClient.java`
- `BaseClientConstants.java`
- `SampleAction.java`
- `SampleCallbackHandler.java`
- `SampleClient.java`
- `TradeResult.java`
- `TraderBean.java`

You will need to look at the examples when reading the information in the following sections.

JAAS and WebLogic Server

The Java Authentication and Authorization Service (JAAS) is a standard extension to the security in the JDK. JAAS provides the ability to enforce access controls based on user identity. WebLogic Server provides JAAS as an alternative to the JNDI authentication mechanism. There are certain considerations when using JAAS authentication.

WebLogic Server clients use the authentication portion of the standard JAAS only. The JAAS LoginContext provides support for the ordered execution of all configured authentication provider LoginModule instances and is responsible for the management of the completion status of each configured provider.

Note the following considerations when using JAAS authentication for Java clients:

- WebLogic Server clients can either use the JNDI login or JAAS login for authentication, however JAAS login is the preferred method.
- While JAAS is the preferred method of authentication, the WebLogic-supplied LoginModule (`weblogic.security.auth.login.UsernamePasswordLoginModule`) only supports

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

username and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client certificate authentication, you must write a custom LoginModule that does certificate authentication.

> **Note**
>
> If you write your own LoginModule for use with WebLogic Server clients, have it call weblogic.security.auth.Authenticate.authenticate() to perform the login.

- To perform a JAAS login from a remote Java client (that is, the Java client is not a WebLogic Server client), you may use the WebLogic-supplied LoginModule to perform the login. However, if you elect not to use the WebLogic-supplied LoginModule but decide to write your own instead, you must have it call the `weblogic.security.auth.Authenticate.authenticate()` method to perform the login.
- If you are using a remote, or perimeter, login system such as Security Assertion Markup Language (SAML), you do not need to call `weblogic.security.auth.Authenticate.authenticate()`. You only need to call the `authenticate()` method if you are using WebLogic Server to perform the logon.

> **Note**
>
> WebLogic Server provides full container support for JAAS authentication and supports full use of JAAS authentication and authorization in application code.

- Within WebLogic Server, JAAS is called to perform the login. Each Authentication provider includes a LoginModule. This is true for servlet logins as well as Java client logins via JNDI or JAAS. The method WebLogic Server calls internally to perform the JAAS logon is `weblogic.security.auth.Authentication.authenticate()`. When using the Authenticate class, `weblogic.security.SimpleCallbackHandler` may be a useful helper class.
- While WebLogic Server does not protect any resources using JAAS authorization (it uses WebLogic security), you can use JAAS authorization in application code to protect the application's own resources.

For more information about JAAS, see the JAAS documentation at http://www.oracle.com/technetwork/java/javase/jaas/index.html.

JAAS Authentication Development Environment

WebLogic Server uses the JAAS classes to reliably and securely authenticate to the server. JAAS implements a Java version of the Pluggable Authentication Module (PAM) framework, which permits applications to remain independent from underlying authentication technologies. Therefore, the PAM framework allows the use of new or updated authentication technologies without requiring modifications to a Java application.

WebLogic Server uses JAAS for remote Java client authentication, and internally for authentication. Therefore, only developers of custom Authentication providers and developers of remote Java client applications need to be involved with JAAS directly. Users of Web browser clients or developers of within-container Java client applications (for example, those calling an EJB from a servlet) do not require direct use or knowledge of JAAS.

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

> **Note**
>
> In order to implement security in a WebLogic client you must install the WebLogic Server software distribution kit on the Java client.

The following topics are covered in this section:

- JAAS Authentication APIs
- JAAS Client Application Components
- WebLogic LoginModule Implementation

JAAS Authentication APIs

To implement Java clients that use JAAS authentication on WebLogic Server, you use a combination of Jakarta EE application programming interfaces (APIs) and WebLogic APIs.

Table 3-1 lists and describes the Java API packages used to implement JAAS authentication. The information in Table 3-1 is taken from the Java API documentation and annotated to add WebLogic Server specific information. For more information on the Java APIs, see the Javadocs at Java SE and JDK API Specification and Jakarta EE Platform API.

Table 3-1 lists and describes the WebLogic APIs used to implement JAAS authentication. See *Java API Reference for Oracle WebLogic Server*.

Table 3-1 Java JAAS APIs

Java JAAS API	Description
`javax.security.auth.Subject` in Java SE and JDK API Specification	The `Subject` class represents the source of the request, and can be an individual user or a group. The `Subject` object is created only after the subject is successfully logged in.
`javax.security.auth.login.LoginContext` in Java SE and JDK API Specification	The `LoginContext` class describes the basic methods used to authenticate `Subjects` and provides a way to develop an application independent of the underlying authentication technology. A `Configuration` specifies the authentication technology, or LoginModule, to be used with a particular application. Therefore, different LoginModules can be plugged in under an application without requiring any modifications to the application itself. After the caller instantiates a `LoginContext`, it invokes the `login` method to authenticate a `Subject`. This `login` method invokes the `login` method from each of the LoginModules configured for the name specified by the caller. If the `login` method returns without throwing an exception, then the overall authentication succeeded. The caller can then retrieve the newly authenticated `Subject` by invoking the `getSubject` method. Principals and credentials associated with the `Subject` may be retrieved by invoking the `Subject`'s respective `getPrincipals`, `getPublicCredentials`, and `getPrivateCredentials` methods. To log the `Subject` out, the caller invokes the `logout` method. As with the `login` method, this `logout` method invokes the `logout` method for each LoginModule configured for this `LoginContext`. For a sample implementation of this class, see Writing a Client Application Using JAAS Authentication.

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

Table 3-1 (Cont.) Java JAAS APIs

Java JAAS API	Description
`javax.security.auth.login.Configuration` in Java SE and JDK API Specification	This is an abstract class for representing the configuration of LoginModules under an application. The `Configuration` specifies which LoginModules should be used for a particular application, and in what order the LoginModules should be invoked. This abstract class needs to be subclassed to provide an implementation which reads and loads the actual configuration. In WebLogic Server, use a login configuration file instead of this class. For a sample configuration file, see Writing a Client Application Using JAAS Authentication. By default, WebLogic Server uses the `Configuration` class, which reads from a configuration file.
`javax.security.auth.spi.LoginModule` in Java SE and JDK API Specification	`LoginModule` describes the interface implemented by authentication technology providers. `LoginModules` are plugged in under applications to provide a particular type of authentication. While application developers write to the `LoginContext` API, authentication technology providers implement the LoginModule interface. A configuration specifies the LoginModule(s) to be used with a particular login application. Therefore, different LoginModules can be plugged in under the application without requiring any modifications to the application itself. **Note:** WebLogic Server provides an implementation of the LoginModule (`weblogic.security.auth.login.UsernamePasswordLoginModule`). Oracle recommends that you use this implementation for JAAS authentication in WebLogic Server Java clients; however, you can develop your own LoginModule.
`javax.security.auth.callback.Callback` in Java SE and JDK API Specification	Implementations of this interface are passed to a `CallbackHandler`, allowing underlying security services to interact with a calling application to retrieve specific authentication data, such as usernames and passwords, or to display information such as error and warning messages. `Callback` implementations do not retrieve or display the information requested by underlying security services. `Callback` implementations simply provide the means to pass such requests to applications, and for applications to return requested information to the underlying security services.
`javax.security.auth.callback.CallbackHandler` in Java SE and JDK API Specification	An application implements a `CallbackHandler` and passes it to underlying security services so that they can interact with the application to retrieve specific authentication data, such as usernames and passwords, or to display information such as error and warning messages. `CallbackHandlers` are implemented in an application-dependent fashion. Underlying security services make requests for different types of information by passing individual `Callbacks` to the `CallbackHandler`. The `CallbackHandler` implementation decides how to retrieve and display information depending on the `Callbacks` passed to it. For example, if the underlying service needs a username and password to authenticate a user, it uses a `NameCallback` and `PasswordCallback`. The `CallbackHandler` can then choose to prompt for a username and password serially, or to prompt for both in a single window.

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

Table 3-2 WebLogic JAAS APIs

WebLogic JAAS API	Description
weblogic.security.auth.Authenticate	An authentication class used to authenticate user credentials. The WebLogic implementation of the LoginModule, (`weblogic.security.auth.login.` `UsernamePasswordLoginModule`, uses this class to authenticate a user and add `Principals` to the `Subject`. Developers who write LoginModules must also use this class for the same purpose.
weblogic.security.auth.callback.ContextHandlerCallback	Underlying security services use this class to instantiate and pass a `ContextHandlerCallback` to the `invokeCallback` method of a `CallbackHandler` to retrieve the ContextHandler related to this security operation. If no ContextHandler is associated with this operation, the `javax.security.auth.callback.UnsupportedCallbackException` is thrown. This callback passes the ContextHandler to LoginModule.login() methods.
weblogic.security.auth.callback.GroupCallback	Underlying security services use this class to instantiate and pass a `GroupCallback` to the `invokeCallback` method of a `CallbackHandler` to retrieve group information.
weblogic.security.auth.callback.URLCallback	Underlying security services use this class to instantiate and pass a `URLCallback` to the `invokeCallback` method of a `CallbackHandler` to retrieve URL information. The WebLogic implementation of the `LoginModule`, (`weblogic.security.auth.login.` `UsernamePasswordLoginModule`, uses this class. **Note:** Application developers should not use this class to retrieve URL information. Instead, they should use the weblogic.security.URLCallbackHandler.
weblogic.security.Security	This class implements the WebLogic Server client `runAs` methods. Client applications use the `runAs` methods to associate their `Subject` identity with the `PrivilegedAction` or `PrivilegedExceptionAction` that they execute. For a sample implementation, see Writing a Client Application Using JAAS Authentication.
weblogic.security.URLCallbackHandler	The class used by application developers for returning a `username`, `password` and `URL`. Application developers should use this class to handle the `URLCallback` to retrieve URL information.

JAAS Client Application Components

At a minimum, a JAAS authentication client application includes the following components:

- Java client

  The Java client instantiates a `LoginContext` object and invokes the login by calling the object's `login()` method. The `login()` method calls methods in each LoginModule to perform the login and authentication.

  The LoginContext also instantiates a new empty `javax.security.auth.Subject` object (which represents the user or service being authenticated), constructs the configured LoginModule, and initializes it with this new `Subject` and `CallbackHandler`.

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

The LoginContext subsequently retrieves the authenticated Subject by calling the LoginContext's `getSubject` method. The LoginContext uses the `weblogic.security.Security.runAs()` method to associate the `Subject` identity with the `PrivilegedAction` or `PrivilegedExceptionAction` to be executed on behalf of the user identity.

- LoginModule

  The LoginModule uses the `CallbackHandler` to obtain the user name and password and determines whether that name and password are the ones required.

  If authentication is successful, the LoginModule populates the Subject with a Principal representing the user. The Principal the LoginModule places in the Subject is an instance of `Principal`, which is a class implementing the `java.security.Principal` interface.

  You can write LoginModule files that perform different types of authentication, including username/password authentication and certificate authentication. A client application can include one LoginModule (the minimum requirement) or several LoginModules.

  > **Note**
  >
  > Use of the JAAS `javax.security.auth.Subject.doAs` methods in WebLogic Server applications do not associate the Subject with the client actions. You can use the `doAs` methods to implement Jakarta EE security in WebLogic Server applications, but such usage is independent of the need to use the `Security.runAs()` method.

- Callbackhandler

  The `CallbackHandler` implements the `javax.security.auth.callback.CallbackHandler` interface. The LoginModule uses the `CallbackHandler` to communicate with the user and obtain the requested information, such as the username and password.

- Configuration file

  This file configures the LoginModule(s) used in the application. It specifies the location of the LoginModule(s) and, if there are multiple LoginModules, the order in which they are executed. This file enables Java applications to remain independent from the authentication technologies, which are defined and implemented using the LoginModule.

- Action file

  This file defines the operations that the client application will perform.

- `ant` build script (`build.xml`)

  This script compiles all the files required for the application and deploys them to the WebLogic Server applications directories.

For a complete working JAAS authentication client that implements the components described here, see the JAAS sample application in *`EXAMPLES_HOME`*`\src\examples\security\jaas`, where *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured and can be found at *`ORACLE_HOME`*`\wlserver\samples\server`. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

For more information on the basics of JAAS authentication, see *JAAS Authentication Tutorial* in Java SE Security Developer's Guide.

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

WebLogic LoginModule Implementation

The WebLogic implementation of the `LoginModule` class (`UsernamePasswordLoginModule.class`) is provided in the WebLogic Server distribution in the `weblogic.jar` file, located in the `WL_HOME\server\lib` directory.

> **Note**
>
> WebLogic Server supports all callback types defined by JAAS as well as all callback types that extend the JAAS specification.

The WebLogic Server `UsernamePasswordLoginModule` checks for existing system user authentication definitions prior to execution, and does nothing if they are already defined.

For more information about implementing JAAS LoginModules, see the *LoginModule Developer's Guide* in Java SE Security Developer's Guide.

JVM-Wide Default User and the runAs() Method

The first time you use the WebLogic Server implementation of the LoginModule (`weblogic.security.auth.login.UsernamePasswordLoginModule`) to log on, the specified user becomes the machine-wide default user for the JVM (Java virtual machine). When you execute the `weblogic.security.Security.runAs()` method, it associates the specified `Subject` with the current thread's access permissions and then executes the action. If a specified `Subject` represents a non-privileged user (users who are not assigned to any groups are considered non-privileged), the JVM-wide default user is used. Therefore, it is important make sure that the `runAs()` method specifies the desired `Subject`. You can do this using one of the following options:

- *Option 1*: If the client has control of `main()`, implement the wrapper code shown in Example 3-1 in the client code.
- *Option 2:* If the client does not have control of `main()`, implement the wrapper code shown in Example 3-1 on each thread's `run()` method.

Example 3-1 runAs() Method Wrapper Code

```
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import weblogic.security.Security;

public class client
{
  public static void main(String[] args)
  {
  Security.runAs(new Subject(),
    new PrivilegedAction() {
      public Object run() {
       //
       //If implementing in client code, main() goes here.
       //
      return null;
      }
    });
```


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

```
  }
}
```

Writing a Client Application Using JAAS Authentication

To use JAAS in a WebLogic Server Java client for authentication, you implement the `LoginModule` and the `CallbackHandler` classes, write a configuration file that specifies which LoginModule classes to use, and perform other tasks.

Perform the following procedure to use JAAS in a WebLogic Server Java client to authenticate a subject:

1. Implement `LoginModule` classes for the authentication mechanisms you want to use with WebLogic Server. You will need a LoginModule class for each type of authentication mechanism. You can have multiple LoginModule classes for a single WebLogic Server deployment.

   > **Note**
   >
   > Oracle recommends that you use the implementation of the LoginModule provided by WebLogic Server (weblogic.security.auth.login.UsernamePasswordLoginModule) for username/password authentication. You can write your own LoginModule for username/password authentication, however, do not attempt to modify the WebLogic Server LoginModule and reuse it. If you write your own LoginModule, you must have it call the weblogic.security.auth.Authenticate.authenticate() method to perform the login. If you use a remote login mechanism such as SAML, you do not need to call the authenticate() method. You only need to call authenticate() if you are using WebLogic Server to perform the logon.

   The `weblogic.security.auth.Authenticate` class uses a JNDI Environment object for initial context as described in Table 3-1.

2. Implement the `CallbackHandler` class that the LoginModule will use to communicate with the user and obtain the requested information, such as the username, password, and URL. The URL can be the URL of a WebLogic cluster, providing the client with the benefits of server failover. The WebLogic Server distribution provides a `SampleCallbackHandler` which is used in the JAAS client sample. The `SampleCallbackHandler.java` code is available as part of the distribution in the directory *`EXAMPLES_HOME`*`\src\examples\security\jaas`. The *`EXAMPLES_HOME`* directory can be found at *`ORACLE_HOME`*`\wlserver\samples\server`.

   > **Note**
   >
   > Instead of implementing your own CallbackHandler class, you can use either of two WebLogic-supplied CallbackHandler classes, weblogic.security.SimpleCallbackHandler or weblogic.security.URLCallbackHandler. For more information on these classes, see *Java API Reference for Oracle WebLogic Server*.

3. Write a configuration file that specifies which LoginModule classes to use for your WebLogic Server and in which order the LoginModule classes should be invoked. See the

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

following sample configuration file used in the JAAS client sample provided in the WebLogic Server distribution.

```
/** Login Configuration for the JAAS Sample Application **/
Sample {
   weblogic.security.auth.login.UsernamePasswordLoginModule
         required debug=false;
};
```

4. In the Java client, write code to instantiate a `LoginContext`. The `LoginContext` consults the configuration file, `sample_jaas.config`, to load the default LoginModule configured for WebLogic Server. See the following sample `LoginContext` instantiation.

```
...
import javax.security.auth.login.LoginContext;
...
    LoginContext loginContext = null;
    try
    {
      // Create LoginContext; specify username/password login module
      loginContext = new LoginContext("Sample",
             new SampleCallbackHandler(username, password, url));
    }
```

> **Note**
>
> If you use another means to authenticate the user, such as an Identity Assertion provider or a remote instance of WebLogic Server, the default LoginModule is determined by the remote source.

5. Invoke the `login()` method of the `LoginContext` instance. The `login()` method invokes all the loaded LoginModules. Each LoginModule attempts to authenticate the subject. If the configured login conditions are not met, the `LoginContext` throws a `LoginException`. See the following example of the `login()` method.

```
...
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.AccountExpiredException;
import javax.security.auth.login.CredentialExpiredException;
...
 /**
     * Attempt authentication
     */
    try
    {
      // If we return without an exception, authentication succeeded
      loginContext.login();
    }
    catch(FailedLoginException fle)
    {
      System.out.println("Authentication Failed, " +
```


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

```
                        fle.getMessage());
      System.exit(-1);
    }
    catch(AccountExpiredException aee)
    {
      System.out.println("Authentication Failed: Account Expired");
      System.exit(-1);
    }
    catch(CredentialExpiredException cee)
    {
      System.out.println("Authentication Failed: Credentials
                        Expired");
      System.exit(-1);
    }
    catch(Exception e)
    {
      System.out.println("Authentication Failed: Unexpected
                        Exception, " + e.getMessage());
      e.printStackTrace();
      System.exit(-1);
    }
```

6. Write code in the Java client to retrieve the authenticated Subject from the `LoginContext` instance using the `javax.security.auth.login.LoginContext.getSubject()` method and call the action as the Subject. Upon successful authentication of a Subject, access controls can be placed upon that Subject by invoking the `weblogic.security.Security.runAs()` method. The `runAs()` method associates the specified Subject with the current thread's access permissions and then executes the action. See the following example implementation of the `getSubject()` and `runAs()` methods.

```
...
/**
 * Retrieve authenticated subject, perform SampleAction as Subject
 */
   Subject subject = loginContext.getSubject();
   SampleAction sampleAction = new SampleAction(url);
   Security.runAs(subject, sampleAction);
   System.exit(0);
...
```

> **Note**
>
> Use of the JAAS `javax.security.auth.Subject.doAs` methods in WebLogic Server applications do not associate the Subject with the client actions. You can use the `doAs` methods to implement Jakarta EE security in WebLogic Server applications, but such usage is independent of the need to use the `Security.runAs()` method.

7. Write code to execute an action if the Subject has the required privileges. Oracle provides a sample implementation, `SampleAction`, of the `java.security.PrivilegedAction` class that executes an EJB to trade stocks. The `SampleAction.java` code is available as part of the distribution in the directory *`EXAMPLES_HOME`*`\src\examples\security\jaas`, where

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

EXAMPLES_HOME represents the directory in which the WebLogic Server code examples are configured, and can be found at *ORACLE_HOME*`\wlserver\samples\server`.

8. Invoke the `logout()` method of the `LoginContext` instance. The `logout()` method closes the user's session and clear the `Subject`. See the following example of the `login()` method.

```
...
import javax.security.auth.login.LoginContext;
...
try
    {
      System.out.println("logging out...");
      loginContext.logout();
    }
```

> **Note**
>
> The LoginModule.logout() method is never called for a WebLogic Authentication provider or a custom Authentication provider, because once the Principals are created and placed into a Subject, the WebLogic Security Framework no longer controls the lifecycle of the Subject. Therefore, code that creates the JAAS LoginContext to log in and obtain the Subject should also call the LoginContext to log out. Calling LoginContext.logout() results in the clearing of the Principals from the Subject.

Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass credentials to WebLogic Server. To do this, a Java client establishes a connection with Oracle WebLogic Server by getting a JNDI `InitialContext` and uses `InitialContext` to look up the resources it needs in the Oracle WebLogic Server JNDI tree.

.

> **Note**
>
> JAAS is the preferred method of authentication, however, the WebLogic Authentication provider's LoginModule supports only user name and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client certificate authentication, you must write a custom Authentication provider whose LoginModule does certificate authentication. For information on how to write LoginModules, see *LoginModule Developer's Guide* in Java SE Security Developer's Guide.

To specify a user and the user's credentials, set the JNDI properties listed in Table 3-1.

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

Table 3-3 JNDI Properties for Authentication

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Provides an entry point into the Oracle WebLogic Server environment. The class weblogic.jndi.WLInitialContextFactory is the JNDI SPI for Oracle WebLogic Server.
`PROVIDER_URL`	Specifies the host and port of the WebLogic Server that provides the name service. For example: `t3s://weblogic:7002`.
`SECURITY_PRINCIPAL`	Specifies the identity of the user when that user authenticates to the default (active) security realm.
`SECURITY_CREDENTIALS`	Specifies the credentials of the user when that user authenticates to the default (active) security realm.

These properties are stored in a hash table that is passed to the `InitialContext` constructor. Example 3-2 illustrates how to use JNDI authentication in a Java client running on WebLogic Server.

> **Note**
>
> For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see JNDI Contexts and Threads and How to Avoid Potential JNDI Context Problems in *Developing JNDI Applications for Oracle WebLogic Server*.

Example 3-2 Example of Authentication

```
...
Hashtable env = new Hashtable();
     env.put(Context.INITIAL_CONTEXT_FACTORY,
               "weblogic.jndi.WLInitialContextFactory");
     env.put(Context.PROVIDER_URL, "t3s://weblogic:7002");
     env.put(Context.SECURITY_PRINCIPAL, "javaclient");
     env.put(Context.SECURITY_CREDENTIALS, "javaclientpassword");
     ctx = new InitialContext(env);
```

Java Client JAAS Authentication Code Examples

The WebLogic Server product provides a complete working JAAS authentication sample.The sample provided by WebLogic Server is located in *EXAMPLES_HOME*`\src\examples\security\jaas`, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured and can be found at *ORACLE_HOME*`\wlserver\samples\server`. For a description of the sample and instructions on how to build, configure, and run this sample, see the `package.html` file in the sample directory. You can modify this code example and reuse it.

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

4

Using SSL Authentication in Java Clients

The Java Secure Socket Extension (JSSE) is a set of packages that support and implement the SSL and TLS protocols. Oracle WebLogic Server provides Secure Sockets Layer (SSL) support for encrypting data transmitted between WebLogic Server clients and servers, Java clients, Web browsers, and other servers. Learn how to implement SSL and digital certificate authentication in Java clients.

- JSSE and WebLogic Server
- Using JNDI Authentication
- SSL Certificate Authentication Development Environment
- Writing Applications that Use SSL
- SSL Client Code Examples

The sections refer to sample code which is optionally included in the WebLogic Server distribution at:

```
EXAMPLES_HOME\src\examples\security\sslclient
```

The *EXAMPLES_HOME* directory can be found at *ORACLE_HOME*`\wlserver\samples\server\examples`.

The `sslclient` directory contains an `instructions.html` file, `ant` build files, and the following Java and JavaServer Pages (`.jsp`) files:

- `MyListener.java`
- `NulledHostnameVerifier.java`
- `NulledTrustManager.java`
- `SSLClient.java`
- `SSLClientServlet.java`
- `SSLSocketClient.java`
- `SnoopServlet.jsp`

You will need to look at the examples when reading the information in the following sections.

JSSE and WebLogic Server

There are certain restrictions when using SSL in WebLogic server-side applications.

The JSSE implementation of WebLogic Server can be used by WebLogic clients, but is not required. Other JSSE implementations can be used for their client-side code outside the server as well.

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

> **Note**
>
> JSSE is the only SSL implementation that is supported. The Certicom-based SSL implementation is removed and is no longer supported in WebLogic Server.

The following restrictions apply when using SSL in WebLogic server-side applications:

- The use of other (third-party) JSSE implementations to develop WebLogic Server applications is not supported. The SSL implementation that WebLogic Server uses is static to the server configuration and is not replaceable by customer applications.
- The WebLogic implementation of JSSE does support JCE Cryptographic Service Providers (CSPs); however, due to the inconsistent provider support for JCE, Oracle cannot guarantee that untested providers will work out of the box. Oracle has tested WebLogic Server with the following providers:
  - The default JCE provider (SunJCE provider). See the *Java Cryptography Architecture (JCA) Reference Guide* and *How to Implement a Provider in the Java Cryptography Architecture* sections in Java SE Security Developer's Guide for information about the SunJCE provider.
  - The Jipher JCE provider. See Using the Jipher JCE Provider in *Administering Security for Oracle WebLogic Server*.

  Other providers may work with WebLogic Server, but an untested provider is not likely to work out of the box. For more information on using the JCE providers supported by WebLogic Server, see Using JCE Providers with WebLogic Server in *Administering Security for Oracle WebLogic Server*.

WebLogic Server uses the HTTPS port for Secure Sockets Layer (SSL) encrypted communication; only SSL can be used on that port.

> **Note**
>
> In order to implement security in a WebLogic client, you must install the WebLogic Server software distribution kit on the Java client.

> **Note**
>
> Although JSSE supports Server Name Indication (SNI) in its SSL implementation, WebLogic Server does not support SNI.

Using JNDI Authentication

Java clients use the Java Naming and Directory Interface (JNDI) to pass credentials to WebLogic Server. A Java client establishes a connection with Oracle WebLogic Server by getting a JNDI `InitialContext`. The Java client then uses the `InitialContext` to look up the resources it needs in the Oracle WebLogic Server JNDI tree.

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

> **Note**
>
> JAAS is the preferred method of authentication; however, the Authentication provider's LoginModule supports only username and password authentication. Thus, for client certificate authentication (also referred to as two-way SSL authentication), you should use JNDI. To use JAAS for client certificate authentication, you must write a custom Authentication provider whose LoginModule does certificate authentication.

To specify a user and the user's credentials, set the JNDI properties listed in Table 4-1.

Table 4-1 JNDI Properties Used for Authentication

Property	Meaning
`INITIAL_CONTEXT_FACTORY`	Provides an entry point into the Oracle WebLogic Server environment. The class weblogic.jndi.WLInitialContextFactory is the JNDI SPI for Oracle WebLogic Server.
`PROVIDER_URL`	Specifies the host and port of the WebLogic Server that provides the name service. For example: `t3s://weblogic:7002`. (t3s is a WebLogic Server proprietary version of SSL.)
`SECURITY_PRINCIPAL`	Specifies the identity of the user when that user authenticates to the default (active) security realm.
`SECURITY_CREDENTIALS`	Specifies the credentials of the user when that user authenticates to the default (active) security realm.

These properties are stored in a hash table which is passed to the `InitialContext` constructor.

Example 4-1 demonstrates how to use one-way SSL certificate authentication in a Java client. For a two-SSL authentication code example, see Example 4-4.

> **Note**
>
> For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see JNDI Contexts and Threads and How to Avoid Potential JNDI Context Problems in *Developing JNDI Applications for Oracle WebLogic Server*.

Example 4-1 Example One-Way SSL Authentication Using JNDI

```
...
Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
              "weblogic.jndi.WLInitialContextFactory");
      env.put(Context.PROVIDER_URL, "t3s://weblogic:7002");
      env.put(Context.SECURITY_PRINCIPAL, "javaclient");
      env.put(Context.SECURITY_CREDENTIALS, "javaclientpassword");
      Context ctx = new InitialContext(env);
```


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

SSL Certificate Authentication Development Environment

To implement SSL authentication in WebLogic Server, you can use a combination of Java application programming interfaces (APIs) and WebLogic APIs. There are certain components of SSL client application such as, HostnameVerifier and TrustManager, that facilitate the implementation of SSL in WebLogic Server.

The following topics are covered in this section:

- SSL Authentication APIs
- SSL Client Application Components

SSL Authentication APIs

To implement Java clients that use SSL authentication on WebLogic Server, use a combination of Java application programming interfaces (APIs) and WebLogic APIs.

Table 4-2 lists and describes the Java APIs packages used to implement certificate authentication. The information in this table is taken from the Java API documentation and annotated to add WebLogic Server specific information. For more information on the Java APIs, see the Javadocs at Java SE and JDK API Specification and Jakarta EE Platform API.

Table 4-3 lists and describes the WebLogic APIs used to implement certificate authentication. See *Java API Reference for Oracle WebLogic Server*.

Table 4-2 Java Certificate APIs

Java Certificate APIs	Description
`javax.crypto` in the Java SE and JDK API Specification	This package provides the classes and interfaces for cryptographic operations. The cryptographic operations defined in this package include encryption, key generation and key agreement, and Message Authentication Code (MAC) generation. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. This package also supports secure streams and sealed objects. Many classes provided in this package are provider-based (see the `java.security.Provider` class). The class itself defines a programming interface to which applications may be written. The implementations themselves may then be written by independent third-party vendors and plugged in seamlessly as needed. Therefore, application developers can take advantage of any number of provider-based implementations without having to add or rewrite code.
`javax.net` in the Java SE and JDK API Specification	This package provides classes for networking applications. These classes include factories for creating sockets. Using socket factories you can encapsulate socket creation and configuration behavior.
`javax.net.SSL` in the Java SE and JDK API Specification	While the classes and interfaces in this package are supported by WebLogic Server, Oracle recommends that you use the `weblogic.security.SSL` package when you use SSL with WebLogic Server.
`java.security.cert` in the Java SE and JDK API Specification	This package provides classes and interfaces for parsing and managing certificates, certificate revocation lists (CRLs), and certification paths. It contains support for X.509 v3 certificates and X.509 v2 CRLs.

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

Table 4-2 (Cont.) Java Certificate APIs

Java Certificate APIs	Description
`java.security.KeyStore` in the Java SE and JDK API Specification	This class represents an in-memory collection of keys and certificates. It is used to manage two types of keystore entries: • Key Entry This type of keystore entry holds cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key. Private keys and certificate chains are used by a given entity for self-authentication. Applications for this authentication include software distribution organizations that sign JAR files as part of releasing and/or licensing software. • Trusted Certificate Entry This type of entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the subject (owner) of the certificate. This type of entry can be used to authenticate other parties.
`java.security.PrivateKey` in the Java SE and JDK API Specification	A private key. This interface contains no methods or constants. It merely serves to group (and provide type safety for) all private key interfaces. **Note:** The specialized private key interfaces extend this interface. For example, see the `DSAPrivateKey` interface in `java.security.interfaces`.
`java.security.Provider` in the Java SE and JDK API Specification	This class represents a "Cryptographic Service Provider" for the Java Security API, where a provider implements some or all parts of Java Security, including: • Algorithms (such as DSA, RSA, MD5 or SHA-1). • Key generation, conversion, and management facilities (such as for algorithm-specific keys). Each provider has a name and a version number, and is configured in each runtime it is installed in. To supply implementations of cryptographic services, a team of developers or a third-party vendor writes the implementation code and creates a subclass of the `Provider` class.
`jakarta.servlet.http.HttpServletRequest` in Jakarta EE Platform API	This interface extends the `ServletRequest` interface to provide request information for HTTP servlets. The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, and so on.).
`jakarta.servlet.http.HttpServletResponse` in Jakarta EE Platform API	This interface extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies. The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, and so on.).
`jakarta.servlet.ServletOutputStream` in Jakarta EE Platform API	This class provides an output stream for sending binary data to the client. A `ServletOutputStream` object is normally retrieved via the `ServletResponse.getOutputStream()` method. This is an abstract class that the servlet container implements. Subclasses of this class must implement the `java.io.OutputStream.write(int)` method.

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

Table 4-2 (Cont.) Java Certificate APIs

Java Certificate APIs	Description
`jakarta.servlet.ServletResponse` in Jakarta EE Platform API	This class defines an object to assist a servlet in sending a response to the client. The servlet container creates a `ServletResponse` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, and so on.).

Table 4-3 WebLogic Certificate APIs

WebLogic Certificate APIs	Description
weblogic.net.http.HttpsURLConnection	This class is used to represent a HTTP with SSL (HTTPS) connection to a remote object. Use this class to make an outbound SSL connection from a WebLogic Server acting as a client to another WebLogic Server.
weblogic.security.SSL.HostnameVerifier	During an SSL handshake, hostname verification establishes that the hostname in the URL matches the hostname in the server's identification; this verification is necessary to prevent man-in-the-middle attacks. WebLogic Server provides a certificate-based implementation of HostnameVerifier which is used by default, and which verifies that the URL hostname matches the CN field value of the server certificate. You can replace the default hostname verifier with a custom hostname verifier using WebLogic Remote Console. This will affect the default for SSL clients running on the server using the WebLogic SSL APIs. In addition, WebLogic SSL APIs such as `HttpsURLConnection`, and `SSLContext` allow the explicit setting of a custom HostnameVerifier.
weblogic.security.SSL.TrustManager	This interface permits the user to override certain validation errors in the peer's certificate chain and allow the handshake to continue. This interface also permits the user to perform additional validation on the peer certificate chain and interrupt the handshake if need be.
weblogic.security.SSL.CertPathTrustManager	This class makes use of the configured CertPathValidation providers to perform extra validation; for example, revocation checking. By default, CertPathTrustManager is installed but configured not to call the CertPathValidators (controlled by the SSLMBean attributes InboundCertificateValidation and OutboundCertificateValidation). Applications that install a custom TrustManager will replace CertPathTrustManager. An application that wants to use a custom TrustManager, and call the CertPathProviders at the same time, can delegate to a CertPathTrustManager from its custom TrustManager.
weblogic.security.SSL.SSLContext	This class holds all of the state information shared across all sockets created under that context.
weblogic.security.SSL.SSLSocketFactory	This class provides the API for creating SSL sockets.
weblogic.security.SSL.SSLValidationConstants	This class defines context element names. SSL performs some built-in validation before it calls one or more CertPathValidator objects to perform additional validation. A validator can reduce the amount of validation it must do by discovering what validation has already been done.

SSL Client Application Components

At a minimum, an SSL client application includes the following components:

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

- Java client

  Typically, a Java client performs these functions:

  - Initializes an `SSLContext` with client identity, trust, a `HostnameVerifier`, and a `TrustManager`.
  - Loads a keystore and retrieves the private key and certificate chain
  - Uses an SSLSocketFactory
  - Uses HTTPS to connect to a JSP served by an instance of WebLogic Server

- HostnameVerifier

  The HostnameVerifier implements the `weblogic.security.SSL.HostnameVerifier` interface.

- HandshakeCompletedListener

  The HandshakeCompletedListener implements the `javax.net.ssl.HandshakeCompletedListener` interface. It is used by the SSL client to receive notifications about the completion of an SSL handshake on a given SSL connection.

- TrustManager

  The TrustManager implements the `weblogic.security.SSL.TrustManager` interface.

For a complete working SSL authentication client that implements the components described here, see the SSLClient sample application in *`EXAMPLES_HOME`*`\src\examples\security\sslclient`, where *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured and can be found at *`ORACLE_HOME`*`\wlserver\samples\server`.

For more information on JSSE authentication, see *Java Secure Socket Extension (JSSE) Reference Guide* in Java SE Security Developer's Guide.

Writing Applications that Use SSL

When you write an application that uses SSL, consider how the application will be used and the special requirements it has for secure communication, such as whether the application is hosted on a WebLogic Server instance acting as a client to another WebLogic Server instance. Other considerations include whether you need to use two-way SSL, a custom host name verifier, a Trust Manager, or other security artifacts.

- Communicating Securely From WebLogic Server to Other WebLogic Servers
- Writing SSL Clients
- Using Two-Way SSL Authentication
- Using a Custom Host Name Verifier
- Using a Trust Manager
- Using an SSLContext
- Using URLs to Make Outbound SSL Connections

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

Communicating Securely From WebLogic Server to Other WebLogic Servers

You can use a URL object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. The `weblogic.net.http.HttpsURLConnection` class provides a way to specify the security context information for a client, including the digital certificate and private key of the client.

The `weblogic.net.http.HttpsURLConnection` class provides methods for determining the negotiated cipher suite, getting/setting a hostname verifier, getting the server's certificate chain, and getting/setting an `SSLSocketFactory` in order to create new SSL sockets.

The SSLClient code example uses the `weblogic.net.http.HttpsURLConnection` class to make an outbound SSL connection. The SSLClient code example is available in the `examples.security.sslclient` package in *`EXAMPLES_HOME`*`\src\examples\security\sslclient`, where *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured and can be found at *`ORACLE_HOME`*`\wlserver\samples\server`.

Writing SSL Clients

This section uses examples to show how to write various types of SSL clients. Examples of the following types of SSL clients are provided:

- SSLClient Sample
- SSLSocketClient Sample
- Using Two-Way SSL Authentication

SSLClient Sample

The SSLClient sample demonstrates how to use the WebLogic SSL library to make outgoing SSL connections using `URL` and `URLConnection` objects. It shows both how to do this from a stand-alone application as well as from a servlet in WebLogic Server.

> **Note**
>
> WebLogic Server acting as an SSL client uses the server's identity certificate for outgoing SSL connections. Applications running on WebLogic Server and using the previously described SSL APIs do not share the server's identity certificates by default, only the trust.

Example 4-2 shows code fragments from the SSLClient example; the complete example is located in the *`EXAMPLES_HOME`*`\src\examples\security\sslclient` directory in the `SSLClient.java` file.

The *`EXAMPLES_HOME`* directory can be found at *`ORACLE_HOME`*`\wlserver\samples\server`.

Example 4-2 SSLClient Sample Code Fragments

```
package examples.security.sslclient;

import java.io.*;
```


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

```
import java.net.URL;
import java.security.Provider;
import jakarta.servlet.ServletOutputStream;
...
  /*
   * This method contains an example of how to use the URL and
   *  URLConnection objects to create a new SSL connection, using
   *  WebLogic SSL client classes.
   */
  public void wlsURLConnect(String host, String port,
                            String sport, String query,
                            OutputStream out)
      throws Exception {
...
      URL wlsUrl = null;
      try {
        wlsUrl = new URL("http", host, Integer.valueOf(port).intValue(),
                         query);
        weblogic.net.http.HttpURLConnection connection =
            new weblogic.net.http.HttpURLConnection(wlsUrl);
        tryConnection(connection, out);
      }
...
      wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(),
                       query);
      weblogic.net.http.HttpsURLConnection sconnection =
          new weblogic.net.http.HttpsURLConnection(wlsUrl);
...
```

SSLSocketClient Sample

The SSLSocketClient sample demonstrates how to use SSL sockets to go directly to the secure port to connect to a JSP served by an instance of WebLogic Server and display the results of that connection. It shows how to implement the following functions:

- Initializing an `SSLContext` with client identity, a `HostnameVerifier`, and a `TrustManager`
- Loading a keystore and retrieving the private key and certificate chain
- Using an `SSLSocketFactory`
- Using HTTPS to connect to a JSP served by WebLogic Server
- Implementing the `javax.net.ssl.HandshakeCompletedListener` interface
- Creating a dummy implementation of the `weblogic.security.SSL.HostnameVerifier` class to verify that the server the example connects to is running on the desired host

Example 4-3 shows code fragments from the SSLSocketClient example; the complete example is located in the *EXAMPLES_HOME*`\src\examples\security\sslclient` directory in the `SSLSocketClient.java` file. (The SSLClientServlet example in the `sslclient` directory is a simple servlet wrapper of the SSLClient example.) The *EXAMPLES_HOME* directory can be found at *ORACLE_HOME*`\wlserver\samples\server`.

Example 4-3 SSLSocketClient Sample Code Fragments

```
package examples.security.sslclient;

import java.io.*;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;
import javax.net.ssl.HandshakeCompletedListener;
```


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

```
import javax.net.ssl.SSLSocket;
import weblogic.security.SSL.HostnameVerifier;
import weblogic.security.SSL.SSLContext;
import weblogic.security.SSL.SSLSocketFactory;
import weblogic.security.SSL.TrustManager;
...
      SSLContext sslCtx = SSLContext.getInstance("https");
      File KeyStoreFile  = new File ("mykeystore");
...
    // Open the keystore, retrieve the private key, and certificate chain
      KeyStore ks = KeyStore.getInstance("jks");
      ks.load(new FileInputStream("mykeystore"), null);
      PrivateKey key = (PrivateKey)ks.getKey("mykey",
                       "testkey".toCharArray());
      Certificate [] certChain = ks.getCertificateChain("mykey");
      sslCtx.loadLocalIdentity(certChain, key);
      HostnameVerifier hVerifier = null;
      if (argv.length < 3)
        hVerifier = new NulledHostnameVerifier();
      else
         hVerifier = (HostnameVerifier)
                     Class.forName(argv[2]).newInstance();

      sslCtx.setHostnameVerifier(hVerifier);
      TrustManager tManager = new NulledTrustManager();
      sslCtx.setTrustManager(tManager);
       System.out.println(" Creating new SSLSocketFactory with SSLContext");
      SSLSocketFactory sslSF = (SSLSocketFactory)
                               sslCtx.getSocketFactory();
      System.out.println(" Creating and opening new SSLSocket with
                           SSLSocketFactory");
      // using createSocket(String hostname, int port)
      SSLSocket sslSock = (SSLSocket) sslSF.createSocket(argv[0],
                           new Integer(argv[1]).intValue());
      System.out.println(" SSLSocket created");
      HandshakeCompletedListener mListener = null;
       mListener = new MyListener();
      sslSock.addHandshakeCompletedListener(new MyListener());
      ...
```

Using Two-Way SSL Authentication

When using certificate authentication, Oracle WebLogic Server sends a digital certificate to the requesting client. The client examines the digital certificate to ensure that it is authentic, has not expired, and matches the Oracle WebLogic Server instance that presented it.

With two-way SSL authentication (a form of mutual authentication), the requesting client also presents a digital certificate to Oracle WebLogic Server. When the instance of WebLogic Server is configured for two-way SSL authentication, requesting clients are required to present digital certificates from a specified set of certificate authorities. Oracle WebLogic Server accepts only digital certificates that are signed by trusted certificate authorities.

For information on how to configure WebLogic Server for two-way SSL authentication, see the Configuring SSL in *Administering Security for Oracle WebLogic Server*.

The following sections describe the different ways two-way SSL authentication can be implemented in WebLogic Server.

- Two-Way SSL Authentication with JNDI
- Using Two-Way SSL Authentication Between WebLogic Server Instances

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

- Using Two-Way SSL Authentication with Servlets

Two-Way SSL Authentication with JNDI

When using JNDI for two-way SSL authentication in a Java client, use the `setSSLContext()` method in the WebLogic JNDI `Environment` class to set the SSLContext onto the current thread for client authentication.

To use `setSSLContext(SSLContext sslctx)`, you pass an SSLContext, with a client certificate created from trustManager and keyManager, to the server using JNDI when the server is configured for two-way SSL. See Class SSLContext in Java SE and JDK API Specification .

> **Note**
>
> Invoking the `setSSLContext` method requires the WebLogic thin T3 client (`wlthint3client.jar`).
>
> `setSSLClientCertificate()` and `setSSLClientKeyPassword()` have been deprecated in this release.

Example 4-4 demonstrates how to use the `setSSLContext()` method for two-way SSL authentication in a Java client.

Example 4-4 Example of a Two-Way SSL Authentication Client That Uses JNDI Environment setSSLContext Method

```
import weblogic.jndi.Environment;

import javax.naming.Context;
import javax.net.ssl.KeyManager;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.SecureRandom;

public class JNDISSLContextClient {
  public static void main(String[] args) throws Exception {
    Context jndiContext = null;
    SSLContext sslContext = null;

    try {
      String identityKeyStore = "path_to_the_identity_keystore";
      String identityKeyStoreType = "type of the identity keystore, e.g. JKS";
      String identityKSPwd = "password_of_the_identity_keystore";
      String alias = "alias_of_identity_certificate_entry";
      String aliasPwd = "pass_word_of_the_alias";
      String trustKeyStore = "path_to_the_trust_keystore";
      String trustKeyStoreType = "type of the trust keystore, e.g. JKS";
      String trustKSPwd = "password_of_the_trust_keystore";
```


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

```
      sslContext = createSSLContext(identityKeyStore, identityKeyStoreType,
identityKSPwd, alias, aliasPwd, trustKeyStore, trustKeyStoreType, trustKSPwd);

      Environment env = new Environment();
      String url = "t3s://localhost:7002";
      env.setProviderUrl(url);
      // The next two set methods are optional if you are using
      // a UserNameMapper interface.
      env.setSecurityPrincipal("system");
      env.setSecurityCredentials("weblogic");

      env.setSSLContext(sslContext);

env.setInitialContextFactory(Environment.DEFAULT_INITIAL_CONTEXT_FACTORY);

      jndiContext = env.getInitialContext();
      Object ejbObj = jndiContext.lookup("ejb");
      // ...
    } finally {
      if (jndiContext != null) jndiContext.close();
    }
  }

  /**
   *
   * @param identityKeyStore the identity keystore, which might contain more
than one entry
   * @param identityKeyStoreType
   * @param identityKeyStorePassword
   * @param identityAlias
   * @param identityAliasPassword
   * @param trustKeyStore
   * @param trustKeyStoreType
   * @param trustKeyStorePassword
   * @param trustKeyStorePassword
   * @return an SSLContext created from the input parameters
   */
  private static SSLContext createSSLContext(String identityKeyStore, String
identityKeyStoreType, String identityKeyStorePassword, String identityAlias,
String identityAliasPassword,
                                             String trustKeyStore, String
trustKeyStoreType, String trustKeyStorePassword) throws Exception {
    //Read the private key and certificate entry under the given alias
    KeyStore identityKS = KeyStore.getInstance(identityKeyStoreType);
    identityKS.load(new java.io.FileInputStream(identityKeyStore),
identityKeyStorePassword.toCharArray());
    KeyStore.Entry entry = identityKS.getEntry(identityAlias, new
KeyStore.PasswordProtection(identityAliasPassword.toCharArray()));

    //KeyStore instance used for the ssl context
    KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
    keystore.load(null, null);

    PrivateKey key = ((KeyStore.PrivateKeyEntry)entry).getPrivateKey();
    java.security.cert.Certificate[] cert =
((KeyStore.PrivateKeyEntry)entry).getCertificateChain();
```


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

```
    byte[] pwd = new byte[10];
    new SecureRandom().nextBytes(pwd);
    char[] entryPassword = new String(pwd).toCharArray();
    byte[] alias = new byte[10];
    new SecureRandom().nextBytes(alias);
    keystore.setKeyEntry(new String(alias), key, entryPassword, cert);
    KeyManagerFactory kmf =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(keystore, entryPassword);
    KeyManager[] keyManagers = kmf.getKeyManagers();

    KeyStore truststore;
    truststore = KeyStore.getInstance(trustKeyStoreType);
    truststore.load(new java.io.FileInputStream(trustKeyStore),
trustKeyStorePassword.toCharArray());
    TrustManagerFactory tmf =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
    tmf.init(truststore);
    TrustManager[] trustManagers = tmf.getTrustManagers();

    SSLContext sslContext = SSLContext.getInstance("TLS");
    sslContext.init(keyManagers, trustManagers, null);

    return sslContext;
  }
}
```

> **Note**
>
> Security provider plug-ins are loaded from the system classpath. The system classpath must specify the implementation of a custom `weblogic.security.providers.authentication.UserNameMapper` interface.

If you have not configured an Identity Assertion provider that performs certificate-based authentication, a Java client running in a JVM with an SSL connection can change the Oracle WebLogic Server user identity by creating a new JNDI `InitialContext` and supplying a new user name and password in the JNDI `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIALS` properties. Any digital certificates passed by the Java client after the SSL connection is made are not used. The new Oracle WebLogic Server user continues to use the SSL connection negotiated with the initial user's digital certificate.

If you have configured an Identity Assertion provider that performs certificate-based authentication, Oracle WebLogic Server passes the digital certificate from the Java client to the class that implements the `UserNameMapper` interface and the `UserNameMapper` class maps the digital certificate to a Oracle WebLogic Server user name. Therefore, if you want to set a new user identity when you use the certificate-based identity assertion, you cannot change the identity. This is because the digital certificate is processed only at the time of the first connection request from the JVM for each `Environment`.

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

> **Note**
>
> Multiple, concurrent, user logins to WebLogic Server from a single client JVM when using two-way SSL and JNDI is not supported. If multiple logins are executed on different threads, the results are undeterminable and might result in one user's requests being executed on another user's login, thereby allowing one user to access another user's data. WebLogic Server does not support multiple, concurrent, certificate-based logins from a single client JVM. For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see JNDI Contexts and Threads and How to Avoid Potential JNDI Context Problems in *Developing JNDI Applications for Oracle WebLogic Server*.

When the JNDI `getInitialContext()` method is called, the Java client and Oracle WebLogic Server execute mutual authentication in the same way that a Web browser performs mutual authentication to get a secure Web server connection. An exception is thrown if the digital certificates cannot be validated or if the Java client's digital certificate cannot be authenticated in the default (active) security realm. The authenticated user object is stored on the Java client's server thread and is used for checking the permissions governing the Java client's access to any protected WebLogic resources.

When you use the WebLogic JNDI `Environment` class, you must create a new `Environment` object for each call to the `getInitialContext()` method. Once you specify a `User` object and security credentials, both the user and their associated credentials remain set in the `Environment` object. If you try to reset them and then call the JNDI `getInitialContext()` method, the original user and credentials are used.

When you use two-way SSL authentication from a Java client, Oracle WebLogic Server gets a unique Java Virtual Machine (JVM) ID for each client JVM so that the connection between the Java client and Oracle WebLogic Server is constant. Unless the connection times out from lack of activity, it persists as long as the JVM for the Java client continues to execute. The only way a Java client can negotiate a new SSL connection reliably is by stopping its JVM and running another instance of the JVM.

The code in Example 4-4 generates a call to the WebLogic Identity Assertion provider that implements the `weblogic.security.providers.authentication.UserNameMapper` interface. The class that implements the `UserNameMapper` interface returns a user object if the digital certificate is valid. Oracle WebLogic Server stores this authenticated user object on the Java client's thread in Oracle WebLogic Server and uses it for subsequent authorization requests when the thread attempts to use WebLogic resources protected by the default (active) security realm.

Writing a User Name Mapper

When using two-way SSL, WebLogic Server verifies the digital certificate of the Web browser or Java client when establishing an SSL connection. However, the digital certificate does not identify the Web browser or Java client as a user in the WebLogic Server security realm. If the Web browser or Java client requests a WebLogic Server resource protected by a security policy, WebLogic Server requires the Web browser or Java client to have an identity. To handle this requirement, the WebLogic Identity Assertion provider allows you to enable a user name mapper that maps the digital certificate of a Web browser or Java client to a user in a WebLogic Server security realm. The user name mapper must be an implementation the `weblogic.security.providers.authentication.UserNameMapper` interface.

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

You have the option of the using the default implementation of the `weblogic.security.providers.authentication.UserNameMapper` interface, `DefaultUserNameMapperImpl`, or developing your own implementation.

The WebLogic Identity Assertion provider can call the implementation of the `UserNameMapper` interface for the following types of identity assertion token types:

- X.509 digital certificates passed via the SSL handshake
- X.509 digital certificates passed via CSIv2
- X.501 distinguished names passed via CSIv2

If you need to map different types of certificates, write your own implementation of the `UserNameMapper` interface.

To implement a `UserNameMapper` interface that maps a digital certificate to a user name, write a `UserNameMapper` class that performs the following operations:

1. Instantiates the `UserNameMapper` implementation class.
2. Creates the `UserNameMapper` interface implementation.
3. Uses the `mapCertificateToUserName()` method to map a certificate to a user name based on a certificate chain presented by the client.
4. Maps a string attribute type to the corresponding `Attribute Value Assertion` field type.

Security provider plug-ins are loaded from the system classpath. The system classpath must specify the implementation of the weblogic.security.providers.authentication.UserNameMapper interface.

Using Two-Way SSL Authentication Between WebLogic Server Instances

You can use two-way SSL authentication in server-to-server communication in which one WebLogic Server instance is acting as the client of another WebLogic Server instance. Using two-way SSL authentication in server-to-server communication enables you to have dependable, highly-secure connections, even without the more common client/server environment.

Example 4-5 shows an example of how to establish a secure connection from a servlet running in one instance of WebLogic Server to a second WebLogic Server instance called `server2.weblogic.com`.

- `setProviderURL`—specifies the URL of the Oracle WebLogic Server instance acting as the SSL server. The WebLogic Server instance acting as SSL client calls this method. The URL specifies the t3s protocol which is a WebLogic Server proprietary protocol built on the SSL protocol. The SSL protocol protects the connection and communication between the two WebLogic Servers instances.
- `setSSLClientCertificate`—specifies the private key and certificate chain to use for the SSL connection. You use this method to specify an input stream array that consists of a private key (which is the first input stream in the array) and a chain of X.509 certificates (which make up the remaining input streams in the array). Each certificate in the chain of certificates is the issuer of the certificate preceding it in the chain.

> **Note**
>
> `setSSLClientCertificate(InputStream[] chain)` is deprecated in this release

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

- `setSSLServerName`—specifies the name of the Oracle WebLogic Server instance acting as the SSL server. When the SSL server presents its digital certificate to the WebLogic Server acting as the SSL client, the name specified using the `setSSLServerName` method is compared to the common name field in the digital certificate. In order for hostname verification to succeed, the names must match. This parameter is used to prevent man-in-the-middle attacks.
- `setSSLRootCAFingerprint`—specifies digital codes that represent a set of trusted certificate authorities, thus specifying trust based on a trusted certificate fingerprint. The root certificate in the certificate chain received from the WebLogic Server instance acting as the SSL server has to match one of the fingerprints specified with this method in order to be trusted. This parameter is used to prevent man-in-the-middle attacks. It provides an addition to the default level of trust, which for clients running on WebLogic Server is that specified by the WebLogic Server trust configuration.

> **Note**
>
> For information on JNDI contexts and threads and how to avoid potential JNDI context problems, see JNDI Contexts and Threads and How to Avoid Potential JNDI Context Problems in *Developing JNDI Applications for Oracle WebLogic Server*.

Example 4-5 Establishing a Secure Connection to Another WebLogic Server Instance

```
FileInputStream [] f = new FileInputStream[3];
   f[0]= new FileInputStream("demokey.pem");
   f[1]= new FileInputStream("democert.pem");
   f[2]= new FileInputStream("ca.pem");
Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");

e.setInitialContextFactory
("weblogic.jndi.WLInitialContextFactory");
Context ctx = new InitialContext(e.getProperties())
```

In Example 4-5, the WebLogic JNDI `Environment` class creates a hash table to store the following parameters:

Using Two-Way SSL Authentication with Servlets

To authenticate Java clients in a servlet (or any other server-side Java class), you must check whether the client presented a digital certificate and if so, whether the certificate was issued by a trusted certificate authority. The servlet developer is responsible for asking whether the Java client has a valid digital certificate. When developing servlets with the WebLogic Servlet API, you must access information about the SSL connection through the `getAttribute()` method of the `HTTPServletRequest` object.

The following attributes are supported in WebLogic Server servlets:

- `jakarta.servlet.request.X509Certificate`
- `java.security.cert.X509Certificate []`—returns an array of the X.509 certificate.
- `jakarta.servlet.request.cipher_suite`—returns a string representing the cipher suite used by HTTPS.

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

- `jakarta.servlet.request.key_size`— returns an integer (0, 40, 56, 128, 168) representing the bit size of the symmetric (bulk encryption) key algorithm.
- `weblogic.servlet.request.SSLSession`
- `javax.net.ssl.SSLSession`—returns the SSL session object that contains the cipher suite and the dates on which the object was created and last used.

You have access to the user information defined in the digital certificates. When you get the `jakarta.servlet.request.X509Certificate` attribute, it is an array of type `java.security.cert.X509Certificate`. You simply cast the array to that type and examine the certificates.

A digital certificate includes information, such as the following:

- The name of the subject (holder, owner) and other identification information required to verify the unique identity of the subject.
- The subject's public key
- The name of the certificate authority that issued the digital certificate
- A serial number
- The validity period (or lifetime) of the digital certificate (as defined by a start date and an end date)

Using a Custom Host Name Verifier

A host name verifier validates that the host to which an SSL connection is made is the intended or authorized party. A host name verifier is useful when a WebLogic client or a WebLogic Server instance is acting as an SSL client to another application server. It helps prevent man-in-the-middle attacks.

> **Note**
>
> Demonstration digital certificates are generated during installation so they do contain the host name of the system on which the WebLogic Server software installed. Therefore, you should leave host name verification on when using the demonstration certificates for development or testing purposes.

By default, WebLogic Server, as a function of the SSL handshake, compares the CN field of the SSL server certificate Subject DN with the host name in the URL used to connect to the server. If these names do not match, the SSL connection is dropped.

The dropping of the SSL connection is caused by the SSL client, which validates the host name of the server against the digital certificate of the server. If anything but the default behavior is desired, you can either turn off host name verification or register a custom host name verifier. Turning off host name verification leaves the SSL connections vulnerable to man-in-the-middle attacks.

You can turn off host name verification in the following ways:

- In WebLogic Remote Console, specify None in the Hostname Verification field that is located on the Advanced Options pane under the SSL tab for the server (for example, `myserver`).
- On the command line of the SSL client, enter the following argument:

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

```
-Dweblogic.security.SSL.ignoreHostnameVerification=true
```

You can write a custom host name verifier. The `weblogic.security.SSL.HostnameVerifier` interface provides a callback mechanism so that implementers of this interface can supply a policy on whether the connection to the URL's host name should be allowed. The policy can be certificate-based or can depend on other authentication schemes.

To use a custom host name verifier, create a class that implements the `weblogic.security.SSL.HostnameVerifier` interface and define the methods that capture information about the server's security identity.

> **Note**
>
> This interface takes new style certificates and replaces the weblogic.security.SSL.HostnameVerifierJSSE interface, which is deprecated.

Before you can use a custom host name verifier, you need to specify the class for your implementation in the following ways:

- In WebLogic Remote Console, set the SSL.HostName Verifier field on the SSL tab under Server: Security configuration to the name of a class that implements this interface. The specified class must have a public no-arg constructor.
- On the command line, enter the following argument:

```
-Dweblogic.security.SSL.hostnameVerifier=hostnameverifier
```

The value for `hostnameverifier` is the name of the class that implements the custom host name verifier.

Example 4-6 shows code fragments from the NulledHostnameVerifier example; the complete example is located in the *EXAMPLES_HOME*`\src\examples\security\sslclient` directory in the `NulledHostnameVerifier.java` file. The *EXAMPLES_HOME* directory can be found at *ORACLE_HOME*`\wlserver\samples\server`. This code example contains a `NulledHostnameVerifier` class which always returns true for the comparison. The sample allows the WebLogic SSL client to connect to any SSL server regardless of the server's host name and digital certificate SubjectDN comparison.

Example 4-6 Hostname Verifier Sample Code Fragment

```
public class NulledHostnameVerifier implements
                    weblogic.security.SSL.HostnameVerifier {
  public boolean verify(String urlHostname, javax.net.ssl.SSLSession session) {
    return true;
  }
}
```

Using a Trust Manager

The `weblogic.security.SSL.TrustManager` interface provides the ability to:

- Ignore specific certificate validation errors
- Perform additional validation on the peer certificate chain

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

> **Note**
>
> This interface takes new style certificates and replaces the weblogic.security.SSL.TrustManagerJSSE interface, which is deprecated.

When an SSL client connects to an instance of WebLogic Server, the server presents its digital certificate chain to the client for authentication. That chain could contain an invalid digital certificate. The SSL specification says that the client should drop the SSL connection upon discovery of an invalid certificate. You can use a custom implementation of the `TrustManager` interface to control when to continue or discontinue an SSL handshake. Using a trust manager, you can ignore certain validation errors, optionally perform custom validation checks, and then decide whether or not to continue the handshake.

Use the `weblogic.security.SSL.TrustManager` interface to create a trust manager. The interface contains a set of error codes for certificate verification. You can also perform additional validation on the peer certificate and interrupt the SSL handshake if need be. After a digital certificate has been verified, the `weblogic.security.SSL.TrustManager` interface uses a callback function to override the result of verifying the digital certificate. You can associate an instance of a trust manager with an SSL context through the `setTrustManager()` method.

You can only set up a trust manger programmatically; its use cannot be defined through WebLogic Remote Console or on the command-line.

> **Note**
>
> Depending on the checks performed, use of a trust manager may potentially impact performance.

Example 4-7 shows code fragments from the NulledTrustManager example; the complete example is located in the *EXAMPLES_HOME*`\src\examples\security\sslclient` directory in the `NulledTrustManager.java` file. The *EXAMPLES_HOME* directory can be found at *ORACLE_HOME*`\wlserver\samples\server`. The SSLSocketClient example uses the custom trust manager. The SSLSocketClient shows how to set up a new SSL connection by using an SSL context with the trust manager.

Example 4-7 NulledTrustManager Sample Code Fragments

```
package examples.security.sslclient;

import weblogic.security.SSL.TrustManager;
import java.security.cert.X509Certificate;
...
public class NulledTrustManager implements TrustManager{
  public boolean certificateCallback(X509Certificate[] o, int validateErr) {
   System.out.println(" --- Do Not Use In Production ---\n" +
                      " By using this NulledTrustManager, the trust in" +
                       " the server's identity is completely lost.\n"
+                      " --------------------------------");
   for (int i=0; i<o.length; i++)
     System.out.println(" certificate " + i + " -- " + o[i].toString());
   return true;
  }
}
```


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

Using the CertPath Trust Manager

The CertPathTrustManager, `weblogic.security.SSL.CertPathTrustManager`, makes use of the default security realm's configured CertPath validation providers to perform extra validation such as revocation checking.

By default, application code using outbound SSL in the server has access only to the built-in SSL certificate validation. However, application code can specify the CertPathTrustManager in order to access any additional certificate validation that the administrator has configured for the server. If you want your application code to also run the CertPath validators, the application code should use the CertPathTrustManager.

There are three ways to use this class:

- The Trust Manager calls the configured CertPathValidators only if the administrator has set a switch on the SSLMBean stating that outbound SSL should use the validators. That is, the application completely delegates validation to whatever the administrator configures. You use the `setUseConfiguredSSLValidation()` method for this purpose. This is the default.
- The Trust Manager always calls any configured CertPathValidators. You use the `setBuiltinSSLValidationAndCertPathValidators()` method for this purpose.
- The Trust Manager never calls any configured CertPathValidators. You use the `setBuiltinSSLValidationOnly()` method for this purpose.

Using a Handshake Completed Listener

The `jakarta.net.ssl.HandshakeCompletedListener` interface defines how an SSL client receives notifications about the completion of an SSL protocol handshake on a given SSL connection. Example 4-8 shows code fragments from the MyListener example; the complete example is located in the *`EXAMPLES_HOME`*`\src\examples\security\sslclient` directory in the `MyListener.java` file. The *`EXAMPLES_HOME`* directory can be found at *`ORACLE_HOME`*`\wlserver\samples\server`.

Example 4-8 MyListener (HandshakeCompletedListener) Sample Code Fragments

```
package examples.security.sslclient;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import javax.net.ssl.HandshakeCompletedListener;
import java.util.Hashtable;
import javax.net.ssl.SSLSession;
...
  public class MyListener implements HandshakeCompletedListener
  {
    public void handshakeCompleted(javax.net.ssl.HandshakeCompletedEvent
                                   event)
    {
      SSLSession session = event.getSession();
      System.out.println("Handshake Completed with peer " +
                         session.getPeerHost());
      System.out.println("   cipher: " + session.getCipherSuite());
      Certificate[] certs = null;
      try
```


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

```
    {
      certs = session.getPeerCertificates();
    }
    catch (SSLPeerUnverifiedException puv)
    {
      certs = null;
    }
    if  (certs != null)
    {
      System.out.println("   peer certificates:");
      for (int z=0; z<certs.length; z++)
            System.out.println("      certs["+z+"]: " + certs[z]);
    }
    else
    {
      System.out.println("No peer certificates presented");
    }
  }
}
```

Using an SSLContext

The `SSLContext` class is used to programmatically configure SSL and to retain SSL session information. Each instance can be configured with the keys, certificate chains, and trusted CA certificates that will be used to perform authentication. SSL sockets created with the same SSLContext and used to connect to the same SSL server could potentially reuse SSL session information. Whether the session information is actually reused depends on the SSL server.

For more information on session caching see SSL Session Behavior in *Administering Security for Oracle WebLogic Server*. To associate an instance of a trust manager class with its SSL context, use the `weblogic.security.SSL.SSLContext.setTrustManager()` method.

You can only set up an SSL context programmatically; you cannot use WebLogic Remote Console or the command line. A Java `new` expression or the `getInstance()` method of the `SSLContext` class can create an `SSLContext` object. The `getInstance()` method is static and it generates a new `SSLContext` object that implements the specified secure socket protocol. An example of using the `SSLContext` class is provided in the `SSLSocketClient.java` sample in *`EXAMPLES_HOME`*`\src\examples\security\sslclient`, where *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured and can be found at *`ORACLE_HOME`*`\wlserver\samples\server`. The SSLSocketClient example shows how to create a new SSL socket factory that will create a new SSL socket using `SSLContext`.

Example 4-9 shows a sample instantiation using the `getInstance()` method.

Example 4-9 SSL Context Code Example

```
import weblogic.security.SSL.SSLContext;
  SSLcontext sslctx = SSLContext.getInstance ("https")
```

Using URLs to Make Outbound SSL Connections

You can use a `URL` object to make an outbound SSL connection from a WebLogic Server instance acting as a client to another WebLogic Server instance. WebLogic Server supports both one-way and two-way SSL authentication for outbound SSL connections.

For one-way SSL authentication, you use the `java.net.URL`, `java.net.URLConnection`, and `java.net.HTTPURLConnection` classes to make outbound SSL connections using `URL` objects. Example 4-10 shows a `simpleURL` class that supports both HTTP and HTTPS URLs and that only uses these Java classes (that is, no WebLogic classes are required). To use the

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

simpleURL class for one-way SSL authentication (HTTPS) on WebLogic Server, all that is required is that "weblogic.net" be defined in the system property for java.protocols.handler.pkgs.

> **Note**
>
> Because the simpleURL sample shown in Example 4-10 defaults trust and hostname checking, this sample requires that you connect to a real Web server that is trusted and that passes hostname checking by default. Otherwise, you must override trust and hostname checking on the command line.

Example 4-10 One-Way SSL Authentication URL Outbound SSL Connection Class That Uses Java Classes Only

```
import java.net.URL;
import java.net.URLConnection;
import java.net.HttpURLConnection;
import java.io.IOException;
public class simpleURL
{
   public static void main (String [] argv)
   {
     if (argv.length != 1)
     {
       System.out.println("Please provide a URL to connect to");
       System.exit(-1);
     }
     setupHandler();
     connectToURL(argv[0]);
   }
   private static void setupHandler()
   {
     java.util.Properties p = System.getProperties();
     String s = p.getProperty("java.protocol.handler.pkgs");
     if (s == null)
       s = "weblogic.net";
     else if (s.indexOf("weblogic.net") == -1)
       s += "|weblogic.net";
     p.put("java.protocol.handler.pkgs", s);
     System.setProperties(p);
   }
   private static void connectToURL(String theURLSpec)
   {
     try
     {
       URL theURL = new URL(theURLSpec);
       URLConnection urlConnection = theURL.openConnection();
       HttpURLConnection connection = null;
       if (!(urlConnection instanceof HttpURLConnection))
       {
         System.out.println("The URL is not using HTTP/HTTPS: " +
                            theURLSpec);
         return;
       }
       connection = (HttpURLConnection) urlConnection;
       connection.connect();
       String responseStr = "\t\t" +
              connection.getResponseCode() + " -- " +
```


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

```
                connection.getResponseMessage() + "\n\t\t" +
                     connection.getContent().getClass().getName() + "\n";
          connection.disconnect();
          System.out.println(responseStr);
        }
        catch (IOException ioe)
        {
          System.out.println("Failure processing URL: " + theURLSpec);
          ioe.printStackTrace();
        }
      }
}
```

For two-way SSL authentication, the `weblogic.net.http.HttpsURLConnection` class provides a way to specify the security context information for a client, including the digital certificate and private key of the client. Instances of this class represent an HTTPS connection to a remote object.

The SSLClient sample code demonstrates using the WebLogic `URL` object to make an outbound SSL connection (see Example 4-11). The code example shown in Example 4-11 is excerpted from the `SSLClient.java` file in the *`EXAMPLES_HOME`*`\src\examples\security\sslclient` directory. The *`EXAMPLES_HOME`* directory can be found at *`ORACLE_HOME`*`\wlserver\samples\server`.

> **Note**
>
> `loadLocalIdentity(InputStream certStream, InputStream keyStream, char[] password)` is deprecated in this release.

Example 4-11 WebLogic Two-Way SSL Authentication URL Outbound SSL Connection Code Example

```
wlsUrl = new URL("https", host, Integer.valueOf(sport).intValue(),
                 query);
weblogic.net.http.HttpsURLConnection sconnection =
         new weblogic.net.http.HttpsURLConnection(wlsUrl);
...
InputStream [] ins = new InputStream[2];
      ins[0] = new FileInputStream("clientkey.pem");
      ins[1] = new FileInputStream("client2certs.pem");
       String pwd = "clientkey";
      sconnection.loadLocalIdentity(ins[0], ins[1], pwd.toCharArray());
```

SSL Client Code Examples

The WebLogic Server product provides a complete working SSL authentication sample.The sample provided by WebLogic Server is located in *`EXAMPLES_HOME`*`\src\examples\security\sslclient`, where *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured and can be found at *`ORACLE_HOME`*`\wlserver\samples\server`. For a description of the sample and instructions on how to build, configure, and run this sample, see the `package.html` file in the sample directory. You can modify this code example and reuse it.

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

5

Securing EJBs

Oracle WebLogic Server supports the Jakarta EE architecture security model for securing EJBs, which includes support for declarative authorization (also referred to in this document as declarative security) and programmatic authorization (also referred to in this document as programmatic security).

- Jakarta EE Architecture Security Model
- Using Declarative Security With EJBs
- EJB Security-Related Deployment Descriptors
- Using Programmatic Security With EJBs

> **Note**
>
> You can use metadata annotations, deployment descriptor files, WebLogic Remote Console, and Jakarta Authorization to secure EJBs. For information on using WebLogic Remote Console to secure EJBs, see Options for Securing Web Application and EJB Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*. For information on Jakarta Authorization, see Using Jakarta Authorization.

Jakarta EE Architecture Security Model

Enterprise tier and web tier applications are made up of components that are deployed into various containers. These components are combined to build a multitier enterprise application. Security for components is provided by their containers. A container provides two kinds of security: declarative and programmatic.

The Jakarta EE Platform includes a Jakarta Security specification that defines portable, plug-in interfaces for authentication and identity stores, and a new injectable-type `SecurityContext` interface that provides an access point for programmatic security. You can use the built-in implementations of these APIs, or define custom implementations.

See Introduction to Security in the Jakarta EE Platform in *The Jakarta EE Tutorial* for complete details about the Jakarta EE security architecture.

Declarative Security

The Jakarta EE Tutorial states that declarative security expresses an application component's security requirements by using either deployment descriptors or annotations.

A deployment descriptor is an XML file that is external to the application and that expresses an application's security structure, including security roles, access control, and authentication requirements.

Annotations, also called metadata, are used to specify information about security within a class file. When the application is deployed, this information can be either used by or overridden by the application deployment descriptor. Annotations save you from having to write declarative

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

information inside XML descriptors. Instead, you simply put annotations on the code, and the required information gets generated. In the tutorial, annotations are used for securing applications wherever possible.

Declarative Authorization Via Annotations

As of EJB 3.x, to make the deployer's task easier, the application developer can define security roles. Developers can specify security metadata annotations directly in the EJB bean class to identify the roles that are allowed to invoke all, or a subset, of the EJB's methods.

As stated in the Securing an Enterprise Bean Using Declarative Security section of *The Jakarta EE Tutorial* , "Declarative security enables the application developer to specify which users are authorized to access which methods of the enterprise beans and to authenticate these users with basic, or user name/password, authentication. Frequently, the person who is developing an enterprise application is not the same person who is responsible for deploying the application. An application developer who uses declarative security to define method permissions and authentication mechanisms is passing along to the deployer a security view of the enterprise beans contained in the EJB JAR. When a security view is passed on to the deployer, he or she uses this information to define method permissions for security roles. If you don't define a security view, the deployer will have to determine what each business method does to determine which users are authorized to call each method."

At deployment time, the deployer then creates these security roles if they do not already exist and maps users to these roles using WebLogic Remote Console to update the security realm. For details, see Security Roles in *Oracle WebLogic Remote Console Online Help*. The deployer can also map any security roles to users using the `weblogic-ejb-jar.xml` deployment descriptor.

> **Note**
>
> Deployment descriptor elements always override their annotation counterparts. In the case of conflicts, the deployment descriptor value overrides the annotation value.
>
> The Jakarta Security specification requires that group principal names are mapped to roles of the same name by default. In WebLogic Server, if the security-role-assignment element in the `weblogic-ejb-jar.xml` deployment descriptor does not declare a mapping between a security role and one or more principals in the WebLogic Server security realm, then the role name is used as the default principal.

Programmatic Security

The Jakarta EE Tutorial states that for an enterprise bean, code embedded in a business method can be used to access a caller's identity programmatically and uses this information to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. The APIs for programmatic security consist of methods of the Jakarta EE `SecurityContext` interface, `EJBContext` interface, and the `HttpServletRequest` interface. These methods allow components to make business-logic decisions based on the security role of the caller or remote user.

The section Securing an Enterprise Bean Programmatically in *The Jakarta EE Tutorial* states that, in general, security management should be enforced by the container in a manner that is transparent to the enterprise bean's business methods. The security APIs described in this section should be used only in the less frequent situations in which the enterprise bean

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

business methods need to access the security context information, such as when you want to restrict access to a particular time of day.

The `SecurityContext` interface, as described in the Jakarta Security specification, defines three methods that allow the bean provider to access security information about the enterprise bean's caller: `getCallerPrincipal`, `getPrincipalsByType`, and `isCallerInRole`.

The `jakarta.ejb.EJBContext` interface provides two methods that allow the bean provider to access security information about the enterprise bean's caller: `getCallerPrincipal()` and `isCallerInRole(String)`.

Note that the newer `SecurityContext` API duplicates some functions of the EJBContext API because it is intended to provide a consistent API across containers. See Using Programmatic Security With EJBs.

Declarative Versus Programmatic Authorization

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application. When choosing the security model that works best for you, consider who is responsible for managing security in your organization. Developers are most familiar with the application components they build, but they might not necessarily be familiar with the deployment environment in which those components run. In addition, as security policies change, it is more economical to reconfigure security declaratively instead of rebuilding, retesting, and redeploying applications, which may be necessary when making programmatic security updates.

As described in Declarative Authorization Via Annotations, to make the deployer's task easier, the application developer can specify security metadata annotations directly in the EJB bean class to identify the roles that are allowed to invoke all, or a subset, of the EJB's methods. However, deployment descriptor elements always override their annotation counterparts, which gives the deployer final control.

Using Declarative Security With EJBs

You can implement declarative security using the security providers using WebLogic Remote Console, or by using Jakarta Authorization. You also use deployment descriptors and metadata annotations for implementing declarative security.

There are three ways to implement declarative security:

1. Security providers using WebLogic Remote Console, as described in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.
2. Jakarta Authorization, as described in Using Jakarta Authorization.
3. Deployment descriptors and metadata annotations, which are discussed in this section.

Which of these three methods is used is defined by the Jakarta Authorization flags and the security model. (Security models are described in Options for Securing EJB and Web Application Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*)

Implementing Declarative Security Via Metadata Annotations

As of EJB 3.0, you are no longer required to create the deployment descriptor files (such as `ejb-jar.xml`). You can now use metadata annotations in the bean file itself to configure metadata.

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

You can still use XML deployment descriptors in addition to, or instead of, the metadata annotations if you so choose.

> **Note**
>
> Deployment descriptor elements always override their annotation counterparts. In the case of conflicts, the deployment descriptor value overrides the annotation value.

To use metadata annotations:

1. Use the metadata annotations feature and create an annotated EJB bean file.
2. At deployment time, the deployer must then create these security roles if they do not already exist and map users to these roles using WebLogic Remote Console to update the security realm. See Security Roles in *Oracle WebLogic Remote Console Online Help*.

The annotations are part of the jakarta.annotation.security package. The following security-related annotations are available:

- jakarta.annotation.security.DeclareRoles — Explicitly lists the security roles that will be used to secure the EJB.
- jakarta.annotation.security.RolesAllowed — Specifies the security roles that are allowed to invoke all the methods of the EJB (when specified at the class-level) or a particular method (when specified at the method-level.)
- jakarta.annotation.security.DenyAll — Specifies that the annotated method can not be invoked by any role.
- jakarta.annotation.security.PermitAll — Specifies that the annotated method can be invoked by all roles.
- jakarta.annotation.security.RunAs — Specifies the role which runs the EJB. By default, the EJB runs as the user who actually invokes it.

Security-Related Annotation Code Examples

The section Securing Access to the EJB in *Developing Jakarta Enterprise Beans for Oracle WebLogic Server* provides an example of a simple stateless session EJB that uses all of the security-related annotations.

The section Specifying Authorized Users by Declaring Security Roles in the *Jakarta EE Tutorial* also discusses how to use annotations to specify the method permissions for the methods of a bean class, with accompanying code examples.

Implementing Declarative Security Via Deployment Descriptors

To implement declarative security in EJBs you can use deployment descriptors (`ejb-jar.xml` and `weblogic-ejb-jar.xml`) to define the security requirements. Example 5-1 shows examples of how to use the `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptors to map security role names to a security realm. The deployment descriptors map the application's logical security requirements to its runtime definitions. And at runtime, the EJB container uses the security definitions to enforce the requirements.

To configure security in the EJB deployment descriptors, perform the following steps (see Example 5-1):

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

1. Use a text editor to create `ejb-jar.xml` and `weblogic-ejb-jar.xml` deployment descriptor files.
2. In the `ejb-jar.xml` file, define the security role name, the EJB name, and the method name.

> **Note**
>
> The proper syntax for a security role name is as defined for an Nmtoken in the Extensible Markup Language (XML) recommendation available on the Web at: http://www.w3.org/TR/REC-xml#NT-Nmtoken.
>
> When specifying security role names, observe the following conventions and restrictions:
>
> - Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
> - Security role names are case sensitive.
> - The suggested convention for security role names is that they be singular.
>
> For more information on configuring security in the ejb-jar.xml file, see the EJB Specification at https://jakarta.ee/specifications/enterprise-beans/.

3. In the WebLogic-specific EJB deployment descriptor file, `weblogic-ejb-jar.xml`, define the security role name and link it to one or more principals (users or groups) in a security realm.

   For more information on configuring security in the `weblogic-ejb-jar.xml` file, see weblogic-ejb-jar.xml Deployment Descriptor Reference in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Example 5-1 Using ejb-jar.xml and weblogic-ejb-jar.xml Files to Map Security Role Names to a Security Realm

```
ejb-jar.xml entries:
          ...
<assembly-descriptor>
  <security-role>
        <role-name>manger</role-name>
  </security-role>
  <security-role>
        <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
          ...
</assembly-descriptor>
           ...
weblogic-ejb-jar.xml entries:
  <security-role-assignment>
      <role-name>manager</role-name>
      <principal-name>al</principal-name>
      <principal-name>george</principal-name>
```


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

```
    <principal-name>ralph</principal-name>
</security-role-assignment>
   ...
```

EJB Security-Related Deployment Descriptors

WebLogic Server supports several deployment descriptor elements that are used in the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files to define security requirements in EJBs.

- ejb-jar.xml Deployment Descriptors
- weblogic-ejb-jar.xml Deployment Descriptors

ejb-jar.xml Deployment Descriptors

The following `ejb-jar.xml` deployment descriptor elements are used to define security requirements in WebLogic Server:

- method
- method-permission
- role-name
- run-as
- security-identity
- security-role
- security-role-ref
- unchecked
- use-caller-identity

method

The `method` element is used to denote a method of an enterprise bean's home or component interface, or, in the case of a message-driven bean, the bean's `onMessage` method, or a set of methods.

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

Table 5-1 method Element

Element	Required/ Optional	Description
`<description>`	Optional	A text description of the method.
`<ejb-name>`	Required	Specifies the name of one of the enterprise beans declared in the `ejb-jar.xml` file.
`<method-intf>`	Optional	Allows you to distinguish between a method with the same signature that is multiply defined across both the home and component interfaces of the enterprise bean.
`<method-name>`	Required	Specifies a name of an enterprise bean method or the asterisk (*) character. The asterisk is used when the element denotes all the methods of an enterprise bean's component and home interfaces.

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

Table 5-1 (Cont.) method Element

Element	Required/ Optional	Description
`<method-params>`	Optional	Contains a list of the fully-qualified Java type names of the method parameters.

Used Within

The `method` element is used within the `method-permission` element.

Example

For an example of how to use the `method` element, see Example 5-1.

method-permission

The `method-permission` element specifies that one or more security roles are allowed to invoke one or more enterprise bean methods. The `method-permission` element consists of an optional description, a list of security role names or an indicator to state that the method is unchecked for authorization, and a list of method elements.

The security roles used in the `method-permission` element must be defined in the `security-role` elements of the deployment descriptor, and the methods must be methods defined in the enterprise bean's component and/or home interfaces.

The following table describes the elements you can define within a `method-permission` element.

Table 5-2 method-permission Element

Element	Required/Optional	Description
`<description>`	Optional	A text description of this security constraint.
`<role-name>` or `<unchecked>`	Required	The `role-name` element or the `unchecked` element must be specified. The `role-name` element contains the name of a security role. The name must conform to the lexical rules for an `NMTOKEN`. The `unchecked` element specifies that a method is not checked for authorization by the container prior to invocation of the method.
`<method>`	Required	Specifies a method of an enterprise bean's home or component interface, or, in the case of a message-driven bean, the bean's `onMessage` method, or a set of methods.

Used Within

The `method-permission` element is used within the `assembly-descriptor` element.

Example

For an example of how to use the `method-permission` element, see Example 5-1.

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

role-name

The role-name element contains the name of a security role. The name must conform to the lexical rules for an `NMTOKEN`.

Used Within

The role-name element is used within the `method-permission`, `run-as`, `security-role, and security-role-ref elements.`

Example

For an example of how to use the `role-name` element, see Example 5-1.

run-as

The `run-as` element specifies the run-as identity to be used for the execution of the enterprise bean. It contains an optional description, and the name of a security role.

Used Within

The run-as element is used within the `security-identity` element.

Example

For an example of how to use the `run-as` element, see Example 5-8.

security-identity

The `security-identity` element specifies whether the caller's security identity is to be used for the execution of the methods of the enterprise bean or whether a specific run-as identity is to be used. It contains an optional description and a specification of the security identity to be used.

The following table describes the elements you can define within an `security-identity` element.

Table 5-3 security-identity Element

Element	Required/Optional	Description
`<description>`	Optional	A text description of the security identity.
`<use-caller-identity> or <run-as>`	Required	The `use-caller-identity` element or the run-as element must be specified. The `use-caller-identity` element specifies that the caller's security identity be used as the security identity for the execution of the enterprise bean's methods. The run-as element specifies the run-as identity to be used for the execution of the enterprise bean. It contains an optional description, and the name of a security role.

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

Used Within

The `security-identity` element is used within the `entity`, `message-driven`, and `session` elements.

Example

For an example of how to use the `security-identity` element, see Example 5-3 and Example 5-8.

security-role

The `security-role` element contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name.

Used Within

The `security-role` element is used within the `assembly-descriptor` element.

Example

For an example of how to use the `assembly-descriptor` element, see Example 5-1.

security-role-ref

The `security-role-ref` element contains the declaration of a security role reference in the enterprise bean's code. The declaration consists of an optional description, the security role name used in the code, and an optional link to a security role. If the security role is not specified, the Deployer must choose an appropriate security role.

The value of the `role-name` element must be the String used as the parameter to the `EJBContext.isCallerInRole(String roleName)` method or the `HttpServletRequest.isUserInRole(String role)` method.

Used Within

The `security-role-ref` element is used within the `entity` and `session` elements.

Example

For an example of how to use the security-role-ref element, see Example 5-2.

Example 5-2 Security-role-ref Element Example

```
<!DOC<weblogic-ejb-jar xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/ns/weblogic/90
http://www.bea.com/ns/weblogic/90/weblogic-ejb-jar.xsd">
<ejb-jar>
 <enterprise-beans>
   ...
   <session>
     <ejb-name>SecuritySLEJB</ejb-name>
     <home>weblogic.ejb20.security.SecuritySLHome</home>
     <remote>weblogic.ejb20.security.SecuritySL</remote>
     <ejb-class>weblogic.ejb20.security.SecuritySLBean</ejb-class>
```


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

```
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <security-role-ref>
       <role-name>rolenamedifffromlink</role-name>
       <role-link>role121SL</role-link>
    </security-role-ref>
    <security-role-ref>
       <role-name>roleForRemotes</role-name>
       <role-link>roleForRemotes</role-link>
    </security-role-ref>
    <security-role-ref>
       <role-name>roleForLocalAndRemote</role-name>
       <role-link>roleForLocalAndRemote</role-link>
    </security-role-ref>
  </session>
   ...
 </enterprise-beans>
</ejb-jar>
```

unchecked

The `unchecked` element specifies that a method is not checked for authorization by the container prior to invocation of the method.

Used Within

The `unchecked` element is used within the `method-permission` element.

Example

For an example of how to use the `unchecked` element, see Example 5-1.

use-caller-identity

The `use-caller-identity` element specifies that the caller's security identity be used as the security identity for the execution of the enterprise bean's methods.

Used Within

The `use-caller-identity` element is used within the `security-identity` element.

Example

For an example of how to use the `use-caller-identity` element, see Example 5-3.

Example 5-3 use-caller-identity Element Example

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SecurityEJB</ejb-name>
      <home>weblogic.ejb20.SecuritySLHome</home>
      <remote>weblogic.ejb20.SecuritySL</remote>
      <local-home>
          weblogic.ejb20.SecurityLocalSLHome
      </local-home>
      <local>weblogic.ejb20.SecurityLocalSL</local>
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <session-type>Stateless</session-type>
```


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

```
      <transaction-type>Container</transaction-type>
    </session>
    <message-driven>
      <ejb-name>SecurityEJB</ejb-name>
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <security-identity>
         <use-caller-identity/>
      </security-identity>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

weblogic-ejb-jar.xml Deployment Descriptors

The following `weblogic-ejb-jar.xml` deployment descriptor elements are used to define security requirements in WebLogic Server:

- client-authentication
- client-cert-authentication
- confidentiality
- externally-defined
- identity-assertion
- iiop-security-descriptor
- integrity
- principal-name
- role-name
- run-as-identity-principal
- run-as-principal-name
- run-as-role-assignment
- security-permission
- security-permission-spec
- security-role-assignment
- transport-requirements

client-authentication

The `client-authentication` element specifies whether the EJB supports or requires client authentication.

The following table defines the possible settings.

Table 5-4 client-authentication Element

Setting	Definition
`none`	Client authentication is not supported.
`supported`	Client authentication is supported, but not required.
`required`	Client authentication is required.

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

Example

For an example of how to use the `client-authentication` element, see Example 5-6.

client-cert-authentication

The `client-cert-authentication` element specifies whether the EJB supports or requires client certificate authentication at the transport level.

The following table defines the possible settings.

Table 5-5 client-cert-authentication Element

Setting	Definition
`none`	Client certificate authentication is not supported.
`supported`	Client certificate authentication is supported, but not required.
`required`	Client certificate authentication is required.

Example

For an example of how to use the `client-cert-authentication` element, see Example 5-10.

confidentiality

The `confidentiality` element specifies the transport confidentiality requirements for the EJB. Using the `confidentiality` element ensures that the data is sent between the client and server in such a way as to prevent other entities from observing the contents.

The following table defines the possible settings.

Table 5-6 confidentiality Element

Setting	Definition
`none`	Confidentiality is not supported.
`supported`	Confidentiality is supported, but not required.
`required`	Confidentiality is required.

Example

For an example of how to use the `confidentiality` element, see Example 5-10 .

externally-defined

The `externally-defined` element lets you explicitly indicate that you want the security roles defined by the `role-name` element in the `weblogic-ejb-jar.xml` deployment descriptors to use the mappings specified in WebLogic Remote Console. The element gives you the flexibility of not having to specify a specific security role mapping for each security role defined in the deployment descriptors for a particular Web application. Therefore, within the same security realm, deployment descriptors can be used to specify and modify security for some applications while WebLogic Remote Console can be used to specify and modify security for others.

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

> **Note**
>
> Starting in version 9.0, the default role mapping behavior is to create empty role mappings when none are specified. In version 8.1, EJB required that role mappings be defined in the weblogic-ejb-jar.xml descriptor or deployment would fail. With 9.0, EJB and WebApp behavior are consistent in creating empty role mappings.
>
> For information on role mapping behavior and backward compatibility settings, see the section Understanding the Combined Role Mapping Enabled Setting in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*. The role mapping behavior for a server depends on which security deployment model is selected in WebLogic Remote Console. For information on security deployment models, see Options for Securing EJB and Web Application Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

When specifying security role names, observe the following conventions and restrictions:

- The proper syntax for a security role name is as defined for an `Nmtoken` in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
- Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
- Security role names are case sensitive.
- The suggested convention for security role names is that they be singular.

Example 5-4 and Example 5-5 show by comparison how to use the `externally-defined` element in the `weblogic-ejb-jar.xml` file. In Example 5-5, the specification of the "manager" `externally-defined` element in the `weblogic-ejb-jar.xml` means that for security to be correctly configured on the `getReceipts` method, the principals for `manager` will have to be created in WebLogic Remote Console.

Example 5-4 Using the ejb-jar.xml and weblogic-ejb-jar.xml Deployment Descriptors to Map Security Roles in EJBs

```
ejb-jar.xml entries:
         ...
<assembly-descriptor>
  <security-role>
        <role-name>manger</role-name>
  </security-role>
  <security-role>
        <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
         ...
</assembly-descriptor>
          ...
weblogic-ejb-jar.xml entries:
  <security-role-assignment>
```


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

```
      <role-name>manager</role-name>
      <principal-name>joe</principal-name>
      <principal-name>Bill</principal-name>
      <principal-name>Mary</principal-name>
      ...
</security-role-assignment>
   ...
```

Example 5-5 Using the externally-defined Element in EJB Deployment Descriptors for Role Mapping

```
ejb-jar.xml entries:
         ...
<assembly-descriptor>
  <security-role>
        <role-name>manger</role-name>
  </security-role>
  <security-role>
        <role-name>east</role-name>
  </security-role>
  <method-permission>
        <role-name>manager</role-name>
        <role-name>east</role-name>
        <method>
          <ejb-name>accountsPayable</ejb-name>
          <method-name>getReceipts</method-name>
        </method>
  </method-permission>
         ...
</assembly-descriptor>
          ...
weblogic-ejb-jar.xml entries:
  <security-role-assignment>
      <role-name>manager</role-name>
      <externally-defined/>
      ...
  </security-role-assignment>
   ...
```

For more information on using WebLogic Remote Console to configure security for EJBs, see Options for Securing EJB and Web Application Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

identity-assertion

The `identity-assertion` element specifies whether the EJB supports identity assertion.

The following table defines the possible settings.

Table 5-7 identity-assertion Element

Setting	Definition
`none`	Identity assertion is not supported
`supported`	Identity assertion is supported, but not required.
`required`	Identity assertion is required.

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

Used Within

The `identity-assertion` element is used with the `iiop-security-descriptor` element.

Example

For an example of how to the `identity-assertion` element, see Example 5-6.

iiop-security-descriptor

The `iiop-security-descriptor` element specifies security configuration parameters at the bean-level. These parameters determine the IIOP security information contained in the interoperable object reference (IOR).

Example

For an example of how to use the `iiop-security-descriptor` element, see Example 5-6.

Example 5-6 iiop-security-descriptor Element Example

```
<weblogic-enterprise-bean>
  <iiop-security-descriptor>
       <transport-requirements>
              <confidentiality>supported</confidentiality>
              <integrity>supported</integrity>
              <client-cert-authorization>
                  supported
              </client-cert-authentication>
       </transport-requirements>
       <client-authentication>supported<client-authentication>
       <identity-assertion>supported</identity-assertion>
  </iiop-security-descriptor>
</weblogic-enterprise-bean>
```

integrity

The `integrity` element specifies the transport integrity requirements for the EJB. Using the integrity element ensures that the data is sent between the client and server in such a way that it cannot be changed in transit.

The following table defines the possible settings.

Table 5-8 integrity Element

Setting	Definition
`none`	Integrity is not supported.
`supported`	Integrity is supported, but not required.
`required`	Integrity is required.

Used Within

The `integrity` element is used within the `transport-requirements` element.

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

Example

For an example of how to use the `integrity` element, see Example 5-10.

principal-name

`The principal-name` element specifies the name of the principal in the WebLogic Server security realm that applies to role name specified in the security-role-assignment element. At least one `principal` is required in the `security-role-assignment` element. You may define more than one `principal-name` for each role name.

> **Note**
>
> If you need to list a significant number of principals, consider specifying groups instead of users. There are performance issues if you specify too many users.

Used Within

The `mk,?"{>L" -name` element is used within the `security-role-assignment` element.

Example

For an example of how to use the `principal-name` element, see Example 5-1.

role-name

The `role-name` element identifies an application role name that the EJB provider placed in the companion `ejb-jar.xml` file. Subsequent principal-name elements in the stanza map WebLogic Server principals to the specified `role-name`.

Used Within

The `role-name` element is used within the `security-role-assignment` element.

Example

For an example of how to use the `role-name` element, see Example 5-1.

run-as-identity-principal

The `run-as-identity-principal` element specifies which security principal name is to be used as the run-as principal for a bean that has specified a security-identity run-as role-name in its ejb-jar deployment descriptor. For an explanation about how run-as role-names are mapped to run-as-identity-principals (or run-as-principal-names, see run-as-role-assignment.

> **Note**
>
> Deprecated: The `run-as-identity-principal` element is deprecated in WebLogic Server 8.1. Use the `run-as-principal-name` element instead.

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

Used Within

The `run-as-identity-principal` element is used within the `run-as-role-assignment` element.

Example

For an example of how to use the `run-as-identity-principal` element, see Example 5-7.

Example 5-7 run-as-identity-principal Element Example

```
ebj-jar.xml:
<ejb-jar>
 <enterprise-beans>
 <session>
   <ejb-name>Caller2EJB</ejb-name>
   <home>weblogic.ejb11.security.CallerBeanHome</home>
   <remote>weblogic.ejb11.security.CallerBeanRemote</remote>
   <ejb-class>weblogic.ejb11.security.CallerBean</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>
   <ejb-ref><ejb-ref-name>Callee2Bean</ejb-ref-name>
     <ejb-ref-type>Session</ejb-ref-type>
     <home>weblogic.ejb11.security.CalleeBeanHome</home>
     <remote>weblogic.ejb11.security.CalleeBeanRemote</remote>
   </ejb-ref>
   <security-role-ref>
     <role-name>users1</role-name>
     <role-link>users1</role-link>
   </security-role-ref>
   <security-identity>
    <run-as>
      <role-name>users2</role-name>
    </run-as>
   </security-identity>
  </session>
 </enterprise-beans>
</ejb-jar>
woblogic-ejb-jar.xml:
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>Caller2EJB</ejb-name>
    <reference-descriptor>
     <ejb-reference-description>
       <ejb-ref-name>Callee2Bean</ejb-ref-name>
        <jndi-name>security.Callee2Bean</jndi-name>
     </ejb-reference-description>
    </reference-descriptor>
    <run-as-identity-principal>wsUser3</run-as-identity-principal>
  </weblogic-enterprise-bean>
  <security-role-assignment>
    <role-name>user</role-name>
    <principal-name>wsUser2</principal-name>
    <principal-name>wsUser3</principal-name>
    <principal-name>wsUser4</principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>
```


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

run-as-principal-name

The `run-as-principal-name` element specifies which security principal name is to be used as the run-as principal for a bean that has specified a security-identity run-as role-name in its ejb-jar deployment descriptor. For an explanation of how the run-as role-names map to run-as-principal-names, see run-as-role-assignment.

Used Within

The `run-as-principal-name` element is used within the `run-as-role-assignment` element.

Example

For an example of how to use the `run-as-principal-name` element, see Example 5-8.

run-as-role-assignment

The `run-as-role-assignment` element is used to map a given security-identity run-as role-name that is specified in the `ejb-jar.xml` file to a `run-as-principal-name` specified in the `weblogic-ejb-jar.xml` file. The value of the `run-as-principal-name` element for a given role-name is scoped to all beans in the `ejb-jar.xml` file that use the specified role-name as their security-identity. The value of the `run-as-principal-name` element specified in `weblogic-ejb-jar.xml` file can be overridden at the individual bean level by specifying a `run-as-principal-name` element under that bean's `weblogic-enterprise-bean` element.

> **Note**
>
> For a given bean, if there is no `run-as-principal-name` element specified in either a `run-as-role-assignment` element or in a bean specific `run-as-principal-name` element, then the EJB container will choose the first principal-name of a security user in the weblogic-enterprise-bean `security-role-assignment` element for the role-name and use that principal-name as the `run-as-principal-name`.

Example

For an example of how to use the run-as-role-assignment element, see Example 5-8.

Example 5-8 run-as-role-assignment Element Example

```
In the ejb-jar.xml file:
// Beans "A_EJB_with_runAs_role_X" and "B_EJB_with_runAs_role_X"
// specify a security-identity run-as role-name "runAs_role_X".
// Bean "C_EJB_with_runAs_role_Y" specifies a security-identity
// run-as role-name "runAs_role_Y".
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SecurityEJB</ejb-name>
      <home>weblogic.ejb20.SecuritySLHome</home>
      <remote>weblogic.ejb20.SecuritySL</remote>
      <local-home>
          weblogic.ejb20.SecurityLocalSLHome
      </local-home>
      <local>weblogic.ejb20.SecurityLocalSL</local>
```


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

```
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    <message-driven>
      <ejb-name>SecurityEJB</ejb-name>
      <ejb-class>weblogic.ejb20.SecuritySLBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <security-identity>
           <run-as>
                 <role-name>runAs_role_X</role-name>
           </run-as>
      </security-identity>
      <security-identity>
            <run-as>
                 <role-name>runAs_role_Y</role-name>
            </run-as>
      </security-identity>
    </message-driven>
  </enterprise-beans>
</ejb-jar>

weblogic-ejb-jar file:

<weblogic-ejb-jar>
   <weblogic-enterprise-bean>
     <ejb-name>A_EJB_with_runAs_role_X</ejb-name>
   </weblogic-enterprise-bean>
   <weblogic-enterprise-bean>
     <ejb-name>B_EJB_with_runAs_role_X</ejb-name>
     <run-as-principal-name>Joe</run-as-principal-name>
   </weblogic-enterprise-bean>
   <weblogic-enterprise-bean>
     <ejb-name>C_EJB_with_runAs_role_Y</ejb-name>
   </weblogic-enterprise-bean>
   <security-role-assignment>
     <role-name>runAs_role_Y</role-name>
     <principal-name>Harry</principal-name>
     <principal-name>John</principal-name>
   </security-role-assignment>
   <run-as-role-assignment>
     <role-name>runAs_role_X</role-name>
     <run-as-principal-name>Fred</run-as-principal-name>
   </run-as-role-assignment>
</weblogic-ejb-jar>
```

Each of the three beans shown in Example 5-8 will choose a different principal name to run as.

- A_EJB_with_runAs_role_X

  This bean's run-as role-name is `runAs_role_X`. The jar-scoped `<run-as-role-assignment>` mapping will be used to look up the name of the principal to use. The `<run-as-role-assignment>` mapping specifies that for <role-name> `runAs_role_X` we are to use `<run-as-principal-name>` `Fred`. Therefore, `Fred` is the principal name that will be used.

- B_EJB_with_runAs_role_X

  This bean's run-as role-name is also `runAs_role_X`. This bean will not use the jar scoped `<run-as-role-assignment>` to look up the name of the principal to use because that value is overridden by this bean's `<weblogic-enterprise-bean>` `<run-as-principal-name>` value `Joe`. Therefore `Joe` is the principal name that will be used.

- C_EJB_with_runAs_role_Y

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

This bean's run-as role-name is `runAs_role_Y`. There is no explicit mapping of `runAs_role_Y` to a run-as principal name, that is, there is no jar scoped `<run-as-role-assignment>` for `runAs_role_Y` nor is there a bean scoped `<run-as-principal-name>` specified in this bean's `<weblogic-enterprise-bean>`. To determine the principal name to use, the `<security-role-assignment>` for `<role-name>` `runAs_role_Y` is examined. The first `<principal-name>` corresponding to a user that is not a Group is chosen. Therefore, `Harry` is the principal name that will be used.

security-permission

The `security-permission` element specifies a security permission that is associated with a Jakarta EE Sandbox.

Example

For an example of how to use the `security-permission` element, see Example 5-9.

security-permission-spec

The `security-permission-spec` element specifies a single security permission based on the Security policy file syntax.

For the implementation of the security permission specification, see *Default Policy Implementation and Policy File Syntax* section in Java SE Security Developer's Guide.

> **Note**
>
> Disregard the optional codebase and signedBy clauses.

Used Within

The `security-permission-spec` element is used within the `security-permission` element.

Example

For an example of how to use the `security-permission-spec` element, see Example 5-9.

Example 5-9 security-permission-spec Element Example

```
<weblogic-ejb-jar>
  <security-permission>
     <description>Optional explanation goes here</description>
     <security-permission-spec>
<!
A single grant statement following the syntax of
http://xmlns.jcp.org/j2se/1.5.0/docs/guide/security/PolicyFiles.html#FileSyntax,
without the codebase and signedBy clauses, goes here. For example:
-->
      grant {
      permission java.net.SocketPermission *, resolve;
      };
     </security-permission-spec>
   </security-permission>
</weblogic-ejb-jar>
```


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

In Example 5-9, `permission java.net.SocketPermission` is the permission class name, "*" represents the target name, and `resolve` (resolve host/IP name service lookups) indicates the action.

security-role-assignment

The `security-role-assignment` element maps application roles in the `ejb-jar.xml` file to the names of security principals available in WebLogic Server.

> **Note**
>
> For information on using the security-role-assignment element in a weblogic-application.xml deployment descriptor for an enterprise application, see Enterprise Application Deployment Descriptor Elements in *Developing Applications for Oracle WebLogic Server*.

Example

For an example of how to use the `security-role-assignment` element, see Example 5-1.

transport-requirements

The `transport-requirements` element defines the transport requirements for the EJB.

Used Within

The `transport-requirements` element is used within the `iiop-security-descriptor` element.

Example

For an example of how to use the `transport-requirements` element, see Example 5-10.

Example 5-10 transport-requirements Element Example

```
<weblogic-enterprise-bean>
  <iiop-security-descriptor>
       <transport-requirements>
              <confidentiality>supported</confidentiality>
              <integrity>supported</integrity>
              <client-cert-authorization>
                   supported
              </client-cert-authentication>
</transport-requirements>
  </iiop-security-descriptor>
<weblogic-enterprise-bean>
```

Using Programmatic Security With EJBs

To implement programmatic security in EJBs, WebLogic Server supports the use of the Jakarta Security `getCallerPrincipal`, `getPrincipalsByType`, and `isCallerInRole` methods of the `SecurityContext` interface, and the `getCallerPrincipal` and `isCallerInRole` methods of the `jakarta.ejb.EJBContext` interface.

The following sections describe these methods in more detail:

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

- SecurityContext Interface Methods
- EJBContext Interface Methods

SecurityContext Interface Methods

The `SecurityContext` interface, as specified in the Jakarta Security specification, defines three methods that allow the bean provider to access security information about the enterprise bean's caller:

- `getCallerPrincipal()` retrieves the Principal that represents the name of the authenticated caller. This is the container-specific representation of the caller principal. The type may differ from the type of the caller principal originally established by an `HttpAuthenticationMechanism`. This method returns null for an unauthenticated caller in either the Servlet Container or the EJB Container. Note that this behavior differs from the behavior of the `EJBContext.getCallerPrincipal()` method, which returns a special principal to represent an anonymous caller.
- `getPrincipalsByType()` retrieves all principals of the given type from the authenticated callers subject. This method returns an empty `Set` if the caller is unauthenticated, or if the requested type is not found.

  Where both a container caller principal and an application caller principal are present, the value returned by `getName()` is the same for both principals.
- `isCallerInRole()` takes a String argument that represents the specific role to be verified. The result must be the same as if the corresponding container-specific call had been made (for example `EJBContext.isCallerInRole()`).

EJBContext Interface Methods

The `jakarta.ejb.EJBContext` interface defines two methods that allow the bean provider to access security information about the enterprise bean's caller:

- `getCallerPrincipal` allows the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to information in a database. This method never returns null. Instead, it returns a principal with a special username to indicate an anonymous/unauthenticated caller. Note that this behavior differs from the behavior of the `SecurityContext.getCallerPrincipal()` method, which returns null for an unauthenticated caller.

  In WebLogic Server, you use the `getCallerPrincipal()` method to determine the caller of the EJB. The `jakarta.ejb.EJBContext.getCallerPrincipal()` method returns a `WLSUser Principal` if one exists in the `Subject` of the calling user. In the case of multiple `WLSUser Principals`, the method returns the first in the ordering defined by the `Subject.getPrincipals().iterator()` method. If there are no `WLSUser Principals`, then the `getCallerPrincipal()` method returns the first non-`WLSGroup Principal`. If there are no `Principals` or all `Principals` are of type `WLSGroup`, this method returns `weblogic.security.WLSPrincipals.getAnonymousUserPrincipal()`. This behavior is similar to the semantics of `weblogic.security.SubjectUtils.getUserPrincipal()` except that `SubjectUtils.getUserPrincipal()` returns a `null` whereas `EJBContext.getCallerPrincipal()`returns `WLSPrincipals.getAnonmyousUserPrincipal()`.
- `isCallerInRole` allows the developer to code the security checks that cannot be easily defined using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database. The enterprise bean code can use the `isCallerInRole` method to test whether the current caller has been

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

assigned to a given security role. Security roles are defined by the bean provider or the application assembler and are assigned by the deployer to principals or principal groups that exist in the operational environment.

In WebLogic Server, the `isCallerInRole()` method is used to determine if the caller (the current user) has been assigned a security role that is authorized to perform actions on the WebLogic resources in that thread of execution. For example, the method `jakarta.ejb.EJBContext.isCallerInRole("admin")`will return `true` if the current user has `admin` privileges.

For information about using these methods, see the Jakarta Enterprise Beans (EJB) specification.

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

6

Using Network Connection Filters

Network connection filters can be used to protect WebLogic resources on individual servers, server clusters, or an entire internal network. Learn how to implement network connection filters in Oracle WebLogic Server.

- The Benefits of Using Network Connection Filters
- Network Connection Filter API
- Guidelines for Writing Connection Filter Rules
- Configuring the WebLogic Connection Filter
- Developing Custom Connection Filters

The Benefits of Using Network Connection Filters

Network connection filters act as a firewall that can be used to allow or deny access to servers in your WebLogic domain based on certain protocols, network addresses and DNS node names.Security roles and security policies let you secure WebLogic resources at the domain level, the application level, and the application-component level. Connection filters let you deny access at the network level. Thus, the network connection filters provide an additional layer of security at the network level. Connection filters can be used to protect server resources on individual servers, server clusters, or an entire internal network.
Connection filters are particularly useful for controlling access through the Administration port. Depending on your network firewall configuration, you might be able to use a connection filter to further restrict administration access. A typical use is to restrict access to the Administration port to only the servers and machines in the domain. Even if an attacker gets access to a machine inside the firewall, they will not be able to perform administration operations unless they are on one of the permitted machines.

Network connection filters are a type of firewall in that they can be configured to filter on protocols, IP addresses, and DNS node names. For example, you can deny any non-SSL connections originating outside of your corporate network. This would ensure that all access from systems on the Internet would be secure.

Network Connection Filter API

Connection filter rules allow you to limit the number of network connections that are accepted. Learn how to create effective connection filter rules and how they are evaluated.

The `weblogic.security.net` API package provides interfaces and classes for developing network connection filters. It also includes a class, `ConnectionFilterImpl`, which is a ready-to-use implementation of a network connection filter. See *Java API Reference for Oracle WebLogic Server* for complete reference information on the network connection filter API.

This section covers the following topics:

- Connection Filter Interfaces
- Connection Filter Classes

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

Connection Filter Interfaces

To implement connection filtering, write a class that implements the connection filter interfaces. The following `weblogic.security.net` interfaces are provided for implementing connection filters:

- ConnectionFilter Interface
- ConnectionFilterRulesListener Interface

ConnectionFilter Interface

This interface defines the `accept()` method, which is used to implement connection filtering. To program the server to perform connection filtering, instantiate a class that implements this interface and then configure that class in WebLogic Remote Console. This interface is the minimum implementation requirement for connection filtering.

> **Note**
>
> Implementing this interface alone does not permit the use of WebLogic Remote Console to enter and modify filtering rules to restrict client connections; you must use some other form (such as a flat file, which is defined in WebLogic Remote Console) for that purpose. To use WebLogic Remote Console to enter and modify filtering rules, you must also implement the ConnectionFilterRulesListener interface. For a description of the ConnectionFilterRulesListener interface, see ConnectionFilterRulesListener Interface.

ConnectionFilterRulesListener Interface

The server uses this interface to determine whether the rules specified in WebLogic Remote Console in the `ConnectionFilterRules` field are valid during startup and at runtime.

> **Note**
>
> You can implement this interface or just use the WebLogic connection filter implementation, weblogic.security.net.ConnectionFilterImpl, which is provided as part of the WebLogic Server product.

This interface defines two methods that are used to implement connection filtering: `setRules()` and `checkRules()`. Implementing this interface in addition to the `ConnectionFilter` interface allows the use of WebLogic Remote Console to enter filtering rules to restrict client connections.

> **Note**
>
> In order to enter and edit connection filtering rules in WebLogic Remote Console, you must implement the ConnectionFilterRulesListener interface; otherwise some other means must be used. For example, you could use a flat file.

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

Connection Filter Classes

Two `weblogic.security.net` classes are provided for implementing connection filters:

- ConnectionFilterImpl Class
- ConnectionEvent Class

ConnectionFilterImpl Class

This class is the WebLogic connection filter implementation of the `ConnectionFilter` and `ConnectionFilterRulesListener` interfaces. Once configured using WebLogic Remote Console, this connection filter accepts all incoming connections by default, and also provides static factory methods that allow the server to obtain the current connection filter. To use this connection to deny access, simply enter connection filter rules using the WebLogic Remote Console.

This class is provided as part of the WebLogic Server product. To configure this class for use, see Configuring the WebLogic Connection Filter.

ConnectionEvent Class

This is the class from which all event state objects are derived. All events are constructed with a reference to the object, that is, the source that is logically deemed to be the object upon which a specific event initially occurred. To create a new `ConnectionEvent` instance, applications use the methods provided by this class: `getLocalAddress()`, `getLocalPort()`, `getRemoteAddress()`, `getRemotePort()`, and `hashcode()`.

Guidelines for Writing Connection Filter Rules

There are certain guidelines for writing connection filter rules. If you do not specify connection rules, then all connections are accepted.

Depending on how you implement connection filtering, connection filter rules can be written in a flat file or input directly on the WebLogic Remote Console.

The following sections provide information and guidelines for writing connection filter rules:

- Connection Filter Rules Syntax
- Types of Connection Filter Rules
- How Connection Filter Rules are Evaluated

Connection Filter Rules Syntax

The syntax of connection filter rules is as follows:

- Each rule must be written on a single line.
- Tokens in a rule are separated by white space.
- A pound sign (#) is the comment character. Everything after a pound sign on a line is ignored.
- Whitespace before or after a rule is ignored.
- Lines consisting only of whitespace or comments are skipped.

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

The format of filter rules differ depending on whether you are using a filter file to enter the filter rules or you enter the filter rules in WebLogic Remote Console.

- When entering the filter rules in WebLogic Remote Console, enter them in the following format:

  ```
  targetAddress localAddress localPort action protocols
  ```

- When specifying rules in the filter file, enter them in the following format:

  ```
  targetAddress action protocols
  ```

  - `targetAddress` specifies one or more systems to filter.
  - `localAddress` defines the host address of the WebLogic Server instance. (If you specify an asterisk (*), the match returns all local IP addresses.)
  - `localPort` defines the port on which the WebLogic Server instance is listening. (If you specify an asterisk (*), the match returns all available ports on the server).
  - `action` specifies the action to perform. This value must be `allow` or `deny`.
  - `protocols` is the list of protocol names to match. The following protocols may be specified: `http`, `https`, `t3`, `t3s`, `ldap`, `ldaps`, `iiop`, `iiops`, and `com`. (Although the `giop`, `giops`, and `dcom` protocol names are still supported, their use is deprecated as of release 9.0; you should use the equivalent `iiop`, `iiops`, and `com` protocol names.)

    > **Note**
    >
    > The SecurityConfigurationMBean provides a CompatibilityConnectionFiltersEnabled attribute for enabling compatibility with previous connection filters.

  - If no protocol is defined, all protocols will match a rule.

Types of Connection Filter Rules

Two types of filter rules are recognized:

- Fast rules

  A fast rule applies to a hostname or IP address with an optional netmask. If a hostname corresponds to multiple IP addresses, multiple rules are generated (in no particular order). Netmasks can be specified either in numeric or dotted-quad form. For example:

  ```
  dialup-555-1212.pa.example.net 127.0.0.1 7001 deny t3 t3s #http(s) OK
  192.168.81.0/255.255.254.0   127.0.0.1 8001 allow   #23-bit netmask
  192.168.0.0/16   127.0.0.1 8002 deny   #like /255.255.0.0
  ```

  Hostnames for fast rules are looked up once at startup of the WebLogic Server instance. While this design greatly reduces overhead at connect time, it can result in the filter obtaining out of date information about what addresses correspond to a hostname. Oracle recommends using numeric IP addresses instead.

- Slow rules

  A slow rule applies to part of a domain name. Because a slow rule requires a connect-time DNS lookup on the client-side in order to perform a match, it may take much longer to run than a fast rule. Slow rules are also subject to DNS spoofing. Slow rules are specified as follows:

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

```
*.script-kiddiez.org 127.0.0.1 7001 deny
```

An asterisk only matches at the head of a pattern. If you specify an asterisk anywhere else in a rule, it is treated as part of the pattern. Note that the pattern will never match a domain name since an asterisk is not a legal part of a domain name.

How Connection Filter Rules are Evaluated

When a client connects to WebLogic Server, the rules are evaluated in the order in which they were written. The first rule to match determines how the connection is treated. If no rules match, the connection is permitted.

To further protect your server and only allow connections from certain addresses, specify the last rule as:

```
0.0.0.0/0  *  *  deny
```

With this as the last rule, only connections that are allowed by preceding rules are allowed, all others are denied. For example, if you specify the following rules:

```
<Remote IP Address> * * allow https
0.0.0.0/0  *  *  deny
```

Only machines with the Remote IP Address are allowed to access the instance of WebLogic Server running connection filter. All other systems are denied access.

> **Note**
>
> The default connection filter implementation interprets a target address of 0 (0.0.0.0/0) as meaning "the rule should apply to all IP addresses." By design, the default filter does not evaluate the port or the local address, just the action. To clearly specify restrictions when using the default filter, modify the rules.
>
> Another option is to implement a custom connection filter.

Configuring the WebLogic Connection Filter

WebLogic Server provides an out-of-the-box network connection filter, which you can configure using the WebLogic Remote Console.

Developing Custom Connection Filters

If you do not want to use the WebLogic connection filter and want to develop you own, you can use the application programming interface (API) provided in the `weblogic.security.net` package to do so.

For a description of the `weblogic.security.net` package, see Network Connection Filter API.

To develop custom connection filters with Oracle WebLogic Server, perform the following steps:

1. Write a class that implements the `ConnectionFilter` interface (minimum requirement).

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

Or, optionally, if you want to use WebLogic Remote Console to enter and modify the connection filtering rules directly, write a class that implements both the `ConnectionFilter` interface and the `ConnectionFilterRulesListener` interface.

2. If you choose the minimum requirement in step 1 (only implementing the `ConnectionFilter` interface), enter the connection filtering rules in a flat file and define the location of the flat file in the class that implements the `ConnectionFilter` interface. Then use WebLogic Remote Console to configure the class in WebLogic Server. For instructions for configuring the class in WebLogic Remote Console, see Using Connection Filters in *Administering Security for Oracle WebLogic Server*.
3. If you choose to implement both interfaces in step 1, use WebLogic Remote Console to configure the class and to enter the connection filtering rules. For instructions on configuring the class inWebLogic Remote Console, see Using Connection Filters in *Administering Security for Oracle WebLogic Server*.

Note that if connection filtering is implemented when a Java or Web browser client tries to connect to a WebLogic Server instance, The WebLogic Server instance constructs a `ConnectionEvent` object and passes it to the `accept()` method of your connection filter class. The connection filter class examines the `ConnectionEvent` object and accepts the connection by returning, or denies the connection by throwing a `FilterException`.

Both implemented classes (the class that implements only the `ConnectionFilter` interface and the class that implements both the `ConnectionFilter` interface and the `ConnectionFilterRulesListener` interface) must call the `accept()` method after gathering information about the client connection. However, if you only implement the `ConnectionFilter` interface, the information gathered includes the remote IP address and the connection protocol: `http`, `https`, `t3`, `t3s`, `ldap`, `ldaps`, `iiop`, `iiops`, or `com`. If you implement both interfaces, the information gathered includes the remote IP address, remote port number, local IP address, local port number and the connection protocol.

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

7

Using Java Security Features to Protect WebLogic Resources

To protect WebLogic resources, Oracle WebLogic Server supports the use of Java security artifacts, such as Jakarta security, Java Security Manager, and Jakarta Authorization.

- Using Jakarta Security to Protect WebLogic Resources
- Using the Java Security Manager to Protect WebLogic Resources
- Using Jakarta Authorization

Using Jakarta Security to Protect WebLogic Resources

You can use Jakarta Security to protect URL (Web), EJBs, and Connector components. Additionally, WebLogic Server extends the connector model of specifying additional security policies in the deployment descriptor to the URL and EJB components.

The connector specification provides for deployment descriptors to specify additional security policies using the `<security-permission>` tag (see Example 7-1):

Example 7-1 Security-Permission Tag Sample

```
<security-permission>
<description> Optional explanation goes here </description>
<security-permission-spec>
<!--
A single grant statement following the syntax of http://xmlns.jcp.org/j2se/1.4.2/docs/
guide/security/PolicyFiles.html#FileSyntax
without the "codebase" and "signedBy" clauses goes here. For example:
-->
grant {
permission java.net.SocketPermission "*", "resolve";
};
</security-permission-spec>
</security-permission>
```

Besides support of the `<security-permission>` tag in the `rar.xml` file, WebLogic Server adds the `<security-permission>` tag to the `weblogic.xml` and `weblogic-ejb-jar.xml` files. This extends the connector model to the two other application types, Web applications and EJBs, provides a uniform interface to security policies across all component types, and anticipates future Jakarta EE specification changes.

> **Note**
>
> Jakarta EE has requirements for Java security default permissions for different application types (see the Jakarta EE specification) as does the Jakarta Connector Architecture specification.

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

Using the Java Security Manager to Protect WebLogic Resources

You can set up the Java Security Manager to be used with WebLogic Server to provide additional protection for resources running in a Java Virtual Machine (JVM). You can also use Printing Security Manager which is an enhancement to the Java Security Manager.

> **Note**
>
> The Java Security Manager and APIs related to it were deprecated in JDK 17 and permanently disabled in JDK 24. There is no direct replacement for the Java Security Manager. See JEP 411 and JEP 486 for discussion and alternatives.

Using a Java Security Manager is an optional security step. The following sections describe how to use the Java Security Manager with WebLogic Server:

- Setting Up the Java Security Manager
- Using Printing Security Manager

For more information on Java Security Manager, see the Java Security Web page at `http://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`.

Setting Up the Java Security Manager

When you run WebLogic Server, WebLogic Server can use the Java Security Manager, which prevents untrusted code from performing actions that are restricted by the Java security policy file.

The JVM has security mechanisms built into it that allow you to define restrictions to code through a Java security policy file. The Java Security Manager uses the Java security policy file to enforce a set of permissions granted to classes. The permissions allow specified classes running in that instance of the JVM to permit or not permit certain runtime operations. In many cases, where the threat model does not include malicious code being run in the JVM, the Java Security Manager is unnecessary. However, when untrusted third-parties use WebLogic Server and untrusted classes are being run, the Java Security Manager may be useful.

To use the Java Security Manager with WebLogic Server, specify the `-Djava.security.policy` and `-Djava.security.manager` arguments when starting WebLogic Server. The `-Djava.security.policy` argument specifies a filename (using a relative or fully-qualified pathname) that contains Java security policies. If you're using Java Security Manager with WebLogic Server, then you must also specify the `-Dweblogic.Name` argument when starting WebLogic Server from the command line using the `java weblogic.Server` command. For example:

```
java -Dweblogic.Name=server-name
      -Djava.security.manager
      -Djava.security.policy[=]=filename
      weblogic.Server
```

WebLogic Server provides a sample Java security policy file located at *`WL_HOME`*`\server\lib\weblogic.policy`. Do not edit this file because it may be overwritten by

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

Patch Set Updates (PSU). Instead, use it as a model to create your own security policy file. Consider concatenating the sample `weblogic.policy` file with your custom security policy file to automatically obtain any updates delivered in the PSU.

> **Note**
>
> This sample policy file is not complete and is not sufficient to start WebLogic Server without modifications. When you create your own custom policy file, make sure that you add various permissions based on your configuration in order for WebLogic Server and all applications to work properly.
>
> Pay particular attention if you apply patches. If you apply patches that include code with system privileges, you may need to make associated changes to any custom Java policy file you are using.

For example, to successfully start WebLogic Server and deploy an application using WebLogic Remote Console, you might need to add permissions such as the following to your custom policy file:

```
permission java.util.PropertyPermission '*', 'read';
permission java.lang.RuntimePermission '*';
permission java.io.FilePermission ' <<ALL FILES>>', 'read,write';
permission javax.management.MBeanPermission '*', '*';
```

If you enable the Java Security Manager but do not specify a security policy file, the Java Security Manager uses the default security policies defined in the `java.policy` file in the `$JAVA_HOME\jre\lib\security` directory.

Define security policies for the Java Security Manager in one of the following ways:

- Modifying your Custom Policy File for General Use
- Setting Application-Type Security Policies
- Setting Application-Specific Security Policies

Modifying your Custom Policy File for General Use

To use the Java Security Manager security policy file with your WebLogic Server deployment, you must specify the location of a custom policy file to the Java Security Manager when you start WebLogic Server. To do this, set the following arguments on the Java command line you use to start the server:

- `java.security.manager` tells the JVM to use a Java security policy file.
- `java.security.policy` tells the JVM the location of the Java security policy file to use. The argument is the fully qualified name of the Java security policy, which in this case, might be `example.policy`.

For example:

```
java...-Djava.security.manager \
   -Djava.security.policy==C:\weblogic\example.policy
```


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

> **Note**
>
> Be sure to use == instead of = when specifying the java.security.policy argument so that only the `example.policy` file is used by the Java Security Manager. The == causes the `example.policy` file to override any default security policy. A single equal sign (=) appends the `example.policy` file to an existing security policy.

If you have extra directories in your `CLASSPATH` or if you are deploying applications in extra directories, add specific permissions for those directories to your `example.policy` file.

Oracle recommends taking the following precautions when using a policy file:

- Do not make your changes to the *`WL_HOME`*`\server\lib\weblogic.policy` file directly as subsequent Patch Set Updates (PSUs) will overwrite your changes. Instead, create a custom policy file based off of *`WL_HOME`*`\server\lib\weblogic.policy` and apply your changes to the custom policy file. Consider configuring your custom policy file to automatically pull in the contents of *`WL_HOME`*`\server\lib\weblogic.policy` so you can get any changes from the PSU automatically.
- Make a backup copy of your custom policy file and put the backup copy in a secure location.
- Set the permissions on the custom policy file via the operating system such that the administrator of the WebLogic Server deployment has write and read privileges and no other users have access to the file.

> **Note**
>
> The Java Security Manager is partially disabled during the booting of Administration and Managed Servers. During the boot sequence, the current Java Security Manager is disabled and replaced with a variation of the Java Security Manager that has the `checkRead()` method disabled. While disabling this method greatly improves the performance of the boot sequence, it also minimally diminishes security. The startup classes for WebLogic Server are run with this partially disabled Java Security Manager and therefore the classes need to be carefully scrutinized for security considerations involving the reading of files.

For more information about the Java Security Manager, see the Javadoc for the `java.lang.SecurityManager` class, available at http://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html.

Setting Application-Type Security Policies

Set default security policies for servlets, EJBs, and Jakarta Connector Architecture resource adapters in the Java security policy file. The default security policies for servlets, EJBs, and resource adapters are defined in the Java security policy file under the following codebases:

- Servlets—`"file:/weblogic/application/defaults/Web"`
- EJBs—`"file:/weblogic/application/defaults/EJB"`
- Resource adapters—`"file:/weblogic/application/defaults/Connector"`

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

> **Note**
>
> These security policies apply to all servlets, EJBs, and resource adapters deployed in the particular instance of WebLogic Server.

Setting Application-Specific Security Policies

Set security policies for a specific servlet, EJB, or resource adapter by adding security policies to their deployment descriptors. Deployment descriptors are defined in the following files:

- Servlets—`weblogic.xml`
- EJBs—`weblogic-ejb-jar.xml`
- Resource adapters—`rar.xml`

> **Note**
>
> The security policies for resource adapters follow the Jakarta EE standard while the security policies for servlets and EJBs follow the WebLogic Server extension to the Jakarta EE standard.

Example 7-2 shows the syntax for adding a security policy to a deployment descriptor:

> **Note**
>
> The <security-permission-spec> tag cannot currently be added to a weblogic-application.xml file, you are limited to using this tag within a weblogic-ejb-jar.xml, rar.xml, or weblogic.xml file. Also, variables are not supported in the <security-permission-spec> attribute.

Example 7-2 Security Policy Syntax

```
<security-permission>
 <description>
  Allow getting the J2EEJ2SETest4 property
 </description>
 <security-permission-spec>
  grant {
    permission java.util.PropertyPermission "welcome.J2EEJ2SETest4","read";
  };
 </security-permission-spec>
</security-permission>
```

Using Printing Security Manager

Printing Security Manager is an enhancement to the Java Security Manager. You can use Printing Security Manager to identify all of the required permissions for any Java application running under Java Security Manager. Unlike The Java Security Manager, which identifies needed permissions one at a time, the Printing Security Manager identifies all of the needed permissions without intervention.

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

For more information on Java Security Manager, see the Java Security Web page at http://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html.

> **Note**
>
> Do not use Printing Security Manager in production environments. It is intended solely for development to identify missing permissions.
>
> It does **not** prevent untrusted code operations.

Printing Security Manager Startup Arguments

To use the Java Security Manager with WebLogic Server, you specify two arguments when starting WebLogic Server:

- `-Djava.security.manager=weblogic.security.psm.PrintingSecurityManager`

  The `-Djava.security.manager` argument tells WebLogic Server which Java Security Manager to start, in this case the Printing Security Manager.

- `-Djava.security.policy`

  The `-Djava.security.policy` argument specifies a file name (using a relative or fully-qualified path name) that contains Java security policies. WebLogic Server provides a sample Java security policy file located at *`WL_HOME`*`\server\lib\weblogic.policy`. Do not use this file directly because it may be overwritten by Patch Set Updates (PSU). Instead, use it as a model to create your own security policy file. Consider concatenating the sample `weblogic.policy` file with your custom security policy file to automatically obtain any updates delivered in the PSU.

  By default, the `startWebLogic` script already includes the `-Djava.security.policy` property, which is set to *`WL_HOME`*`/server/lib/weblogic.policy`. Update the `-Djava.security.policy` to specify the file location of your custom policy file.

> **Note**
>
> The sample policy file at *`WL_HOME`*`\server\lib\weblogic.policy` is not complete and is not sufficient to start WebLogic Server without first being modified. When you create your own custom policy file, make sure that you add various permissions based on your configuration in order for WebLogic Server and all applications to work properly.
>
> See the following sections:
>
> - Modifying your Custom Policy File for General Use
> - Setting Application-Type Security Policies
> - Setting Application-Specific Security Policies

Starting WebLogic Server With Printing Security Manager

To start WebLogic Server with the Printing Security Manager from a UNIX shell, pass the following argument to the startWebLogic.sh script in *DOMAIN_HOME*. This example uses the default weblogic.policy Java policy file from startWeblogic.sh.

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

```
startWeblogic.sh
-Djava.security.manager=weblogic.security.psm.PrintingSecurityManager
```

For a Windows system without a UNIX shell, first set the startup options in *JAVA_OPTIONS*, and then use the `startWebLogic.cmd` script in *DOMAIN_HOME* to start WebLogic Server. This example uses the default weblogic.policy Java policy file from `startWeblogic.cmd`.

```
$ set JAVA_OPTIONS=-Djava.security.manager=weblogic.security.psm.PrintingSecurityManager

$ DOMAIN_HOME\startWeblogic.cmd
```

Writing Output Files

Printing Security Manager generates output that lists which code source needs which permissions. It also generates a policy grant that you can copy and paste into the policy file.

By default, output is to System.out. You can configure output files via two arguments:

- `-Doracle.weblogic.security.manager.printing.file=psm_perms.txt`
- `-Doracle.weblogic.security.manager.printing.generated.grants.file=psm_grants.txt`

The value of the first system argument is a file to which Printing Security Manager writes all missing-permission messages. The value of the second argument is a file to which Printing Security Manager writes the missing policy grants.

For example, for a Windows system without a UNIX shell, add the argument to *JAVA_OPTIONS*:

```
$ set JAVA_OPTIONS=-Djava.security.manager=weblogic.security.psm.PrintingSecurityManager
-Doracle.weblogic.security.manager.printing.file=psm_perms.txt

$ startWeblogic.cmd
```

If you do not specify the full path for the output files, they are created in *DOMAIN_HOME*.

Using Jakarta Authorization

Jakarta Authorization provides an alternate authorization mechanism for the EJB and servlet containers in a WebLogic Server domain. You can enable the WebLogic JACC provider by specifying certain system property-value pairs.

The Jakarta Authorization specification is part of the Jakarta EE Platform. Jakarta Authorization extends the Java permission-based security model to EJBs and servlets. Jakarta Authorization is defined at https://jakarta.ee/specifications/authorization/. Jakarta Authorization was previously known as Java Authorization Contract for Containers (JACC).

As shown in Table 7-2, when Jakarta Authorization is configured, the WebLogic Security framework access decisions, adjudication, and role mapping functions are not used for EJB and servlet authorization decisions.

WebLogic Server implements a Jakarta Authorization provider which, although fully compliant with the Jakarta Authorization specification, is not as optimized as the WebLogic Authorization provider. The Jakarta Authorization classes are used for rendering access decisions. Because the Jakarta Authorization specification does not define how to address role mapping, WebLogic JACC classes are used for role-to-principal mapping.

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

> **Note**
>
> The Jakarta Authorization classes used by WebLogic Server do not include an implementation of a Policy object for rendering decisions but instead rely on the `java.security.Policy` object (see Java SE and JDK API Specification).

This section discusses the following topics:

- Comparing the WebLogic JACC Provider with the WebLogic Authorization Provider
- Enabling the WebLogic JACC Provider

Table 7-2 shows which providers are used for role mapping when JACC is enabled.

Table 7-1 When Jakarta Authorization is Enabled

Status	Provider used for EJB/ Servlet Authorization and Role Mapping	Provider used for all other Authorization and Role Mapping	EJB/Servlet Roles and Policies Can be Viewed and Modified by WebLogic Remote Console
Jakarta Authorization is enabled	JACC provider	WebLogic Security Framework providers	No
Jakarta Authorization is not enabled	WebLogic Security Framework providers	WebLogic Security Framework providers	Yes, depending on settings

> **Note**
>
> In a domain, either enable Jakarta Authorization on all servers or on none. The reason is that Jakarta Authorization is server-specific, while the WebLogic Security Framework is realm/domain specific. If you enable Jakarta Authorization, either by using the WebLogic JACC provider or (recommended) by creating your own Jakarta Authorization provider, you are responsible for keeping EJB and servlet authorization policies synchronized across the domain. For example, applications are redeployed each time a server boots. If a server configured for Jakarta Authorization reboots without specifying the Jakarta Authorization options on the command line, the server uses the default WebLogic Authorization provider for EJB and servlet role mapping and authorization decisions.

Comparing the WebLogic JACC Provider with the WebLogic Authorization Provider

The WebLogic JACC provider fully complies with the Jakarta Authorization specification; however, it does not support dynamic role mapping, nor does it address authorization decisions for resources other than EJBs and servlets. For better performance, and for more flexibility regarding security features, Oracle recommends using SSPI-based providers.

Table 7-2 compares the features provided by the WebLogic JACC provider with those of the WebLogic Authorization provider.

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

Table 7-2 Comparing the WebLogic JACC Provider with the WebLogic Authorization Provider

WebLogic JACC Provider	WebLogic Authorization Provider
Implements the Jakarta Authorization specification	Value-added security framework
Addresses only EJB and servlet deployment/ authorization decisions	Addresses deployment/authorization decisions
Uses the `java.security.Policy` object to render decisions	Allows for multiple authorization/role providers
Static role mapping at deployment time	Dynamic role mapping
Jakarta EE permissions control access	Entitlements engine controls access
Role and role-to-principal mappings are modifiable only through deployment descriptors	Roles and role-to-principal mappings are modifiable through deployment descriptors and WebLogic Remote Console

Enabling the WebLogic JACC Provider

In the command that starts WebLogic Server, you can enable the WebLogic JACC provider by specifying the following system property/value pairs:

- **Property:**

  `jakarta.security.jacc.PolicyConfigurationFactory.provider`

  Value:

  `weblogic.security.jacc.simpleprovider.PolicyConfigurationFactoryImpl`

- **Property:**

  `jakarta.security.jacc.policy.provider`

  Value:

  `weblogic.security.jacc.simpleprovider.SimpleJACCPolicy`

- **Property:**

  `weblogic.security.jacc.RoleMapperFactory.provider`

  Value:

  `weblogic.security.jacc.simpleprovider.RoleMapperFactoryImpl`

> **Note**
>
> If the system properties, `-Djakarta.security.jacc.PolicyConfigurationFactory.provider` and `-Djakarta.security.jacc.policy.provider` are specified, then WebLogic Server automatically initializes the default `RoleMapperFactory` property. Therefore, you do not need to specify the `weblogic.security.jacc.RoleMapperFactory.provider` system property to enable the WebLogic JACC provider.

For example, assuming a properly configured `weblogic.policy` file, the following command line enables the WebLogic JACC provider:

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

```
# ./startWebLogic.sh -Djakarta.security.jacc.policy.provider=\
weblogic.security.jacc.simpleprovider.SimpleJACCPolicy \
-Djakarta.security.jacc.PolicyConfigurationFactory.provider=\
weblogic.security.jacc.simpleprovider.PolicyConfigurationFactoryImpl \
```


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

8

SAML APIs

Oracle WebLogic Server supports the use of Security Assertion Markup Language (SAML) APIs. SAML is an XML-based protocol for exchanging security information between software entities on the Web. SAML security is based on the interaction of asserting and relying parties.SAML provides single sign-on capabilities; users can authenticate at one location and then access service providers at other locations without having to log in multiple times. WebLogic Server supports SAML 2.0. The WebLogic Server implementation:

- Supports the HTTP POST, HTTP Artifact, and HTTP Redirect bindings for the Web SSO profile for SAML 2.0.
- Supports SAML authentication and attribute statements (does not support SAML authorization statements)

For a general description of SAML and SAML assertions in a WebLogic Server environment, see Security Assertion Markup Language (SAML) in *Understanding Security for Oracle WebLogic Server*.

For information on configuring a SAML credential mapping provider, see Configuring a SAML 2.0 Credential Mapping Provider for SAML 2.0 in *Administering Security for Oracle WebLogic Server*.

For access to the SAML specifications, go to http://www.oasis-open.org. Also see the Security Assertion Markup Language (SAML) 2.0 Technical Overview (http://www.oasis-open.org/committees/download.php/11511/sstc-saml-tech-overview-2.0-draft-03.pdf).

This chapter includes the following sections:

- SAML API Description
- Configuring SAML SSO Attribute Support

SAML API Description

Learn about the WebLogic SAML APIs that you can use to implement SAML in WebLogic Server.

Table 8-1 lists the WebLogic SAML 2.0 APIs. See the Javadoc for details.

Table 8-1 WebLogic SAML 2.0 APIs

WebLogic SAML 2.0 APIs	Description
com.bea.security.saml2.providers	Provides interfaces and classes for the configuration, control, and monitoring of SAML 2.0 security providers in a WebLogic security realm.
SAML2AttributeInfo	A class that represents a single attribute of a SAML 2.0 Assertion AttributeStatement.
SAML2AttributeStatementInfo	A class that represents an AttributeStatement in a SAML 2.0 Assertion.
SAML2CredentialAttributeMapper	Interface used to perform mapping from Subject to SAML 2.0 Assertion attributes.

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

Table 8-1 (Cont.) WebLogic SAML 2.0 APIs

WebLogic SAML 2.0 APIs	Description
SAML2CredentialNameMapper	Interface used to perform the mapping of user and group information to SAML 2.0 assertions.
SAML2IdentityAsserterAttributeMapper	Interface used to perform mapping from SAML 2.0 Attribute Statement to Attribute Principals.
SAML2IdentityAsserterNameMapper	Interface used to perform the mapping of user information contained in a SAML 2.0 assertion to a local user name.
SAML2NameMapperInfo	The SAML2NameMapperInfo is used to represent user name and group information contained in SAML 2.0 assertions.
com.bea.security.saml2.providers.registry	Abstract interfaces for SAML 2.0 Identity Provider and Service Provider partners and metadata.
BindingClientPartner	Binding Client partner is a partner that supports backend channel communication.
IdPPartner	Abstract representation of a SAML 2.0 Identity Provider partner.
Endpoint	Abstract representation of a SAML 2.0 service endpoint.
IndexedEndpoint	This class represents the end point that could be indexed, like Artifact Resolution Service's end point.
MetadataPartner	Metadata partner contains contact information for the partner, which is mainly required by the SAML 2.0 metadata profile.
Partner	Abstract representation of a SAML 2.0 partner. This interface defines mandatory information for a partner.
SPPartner	Abstract representation of a SAML 2.0 Service Provider partner.
WebSSOIdPPartner	Abstract representation of a SAML 2.0 Identity Provider partner for Web SSO profile.
WebSSOPartner	Abstract representation of a SAML 2.0 partner for Web SSO profile.
WebSSOSPPartner	Abstract representation of a SAML 2.0 Service Provider partner for Web SSO profile.
WSSIdPPartner	Abstract representation of a SAML 2.0 Identity Provider partner for WSS SAML Token profile.
WSSPartner	Abstract representation of a SAML 2.0 partner for WSS SAML Token profile.
WSSSPPartner	Abstract representation of a SAML 2.0 Service Provider partner for WSS SAML Token profile. It has no specific attributes/methods.

Configuring SAML SSO Attribute Support

A SAML assertion is a piece of data produced by a SAML authority regarding either an act of authentication performed on a subject, attribute information about the subject, or authorization data applying to the subject with respect to a specified resource. You can configure SAML SSO attributes to be used with SAML 2.0.

This section describes the following topics:

- What Are SAML SSO Attributes?
- APIs for SAML Attributes
- SAML 2.0 Basic Attribute Profile Required

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

- Passing Multiple Attributes to SAML Credential Mappers
- How to Implement SAML Attributes
- Examples of the SAML 2.0 Attribute Interfaces
- Make the Custom SAML Credential Attribute Mapper Class Available in the Console
- Make the Custom SAML Identity Asserter Class Available in the Console

What Are SAML SSO Attributes?

The SAML specification (see `http://www.oasis-open.org`) allows additional, unspecified information about a particular subject to be exchanged between SAML partners as attribute statements in an assertion. A **SAML attribute assertion** is therefore a particular type of SAML assertion that conveys site-determined information about attributes of a Subject.

The SAML 2.0 Credential Mapping provider and Identity Assertion provider mechanisms support the use of a custom attribute mapper that can obtain additional attributes (other than group information) to be written into SAML assertions, and to then map attributes from incoming SAML assertions.

To do this:

- The SAML credential mapper (on the SAML Identity Provider site) determines how to package the attributes based on the existence of this custom attribute mapper.
- The SAML identity asserter (on the SAML Service Provider site) determines how to get the attributes based on the configuration of the custom name mapper.
- The Java Subject is used to make the attributes extracted from assertions available to applications. This requires that the SAML Authentication provider be configured and the virtual user be enabled on a SAML partner.

APIs for SAML Attributes

Use the following SAML 2.0 attribute APIs:

- `SAML2AttributeInfo`
- `SAML2AttributeStatementInfo`
- `SAML2CredentialAttributeMapper`
- `SAML2IdentityAsserterAttributeMapper`

Subsequent sections describe the use of these SAML attribute APIs.

SAML 2.0 Basic Attribute Profile Required

SAML 1.1 does not prescribe the name format of the SAML attribute.

However, only the SAML 2.0 Basic Attribute Profile is supported for SAML 2.0. Only attributes with the `urn:oasis:names:tc:SAML:2.0:attrname-format:basic` name format in `SAML2AttributeInfo` are written into a SAML 2.0 assertion.

The `urn:oasis:names:tc:SAML:2.0:attrname-format:basic` name format is the default, so you need not set it.

If you do set the name format, you must specify `urn:oasis:names:tc:SAML:2.0:attrname-format:basic` in the `SAML2.AttributeInfo.setAttributeNameFormat` method, as follows:

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

```
SAML2AttributeInfo attrInfo = new SAML2AttributeInfo(
"AttributeWithSingleValue", "ValueOfAttributeWithSingleValue");
attrInfo.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-format:basic");
attrs.add(attrInfo);
```

Passing Multiple Attributes to SAML Credential Mappers

When the configured attribute mapper is called, it returns `Collection<SAML2AttributeStatementInfo>`. You can specify multiple attribute statements, each containing multiple attributes, each possibly having multiple attribute values.

An example of doing this is as follows:

```
private Collection<SAML2AttributeStatementInfo> getAttributeStatementInfo(
Subject subject, ContextHandler handlers) {
Collection<SAML2AttributeInfo> attrs = new ArrayList<SAML2AttributeInfo>();

SAML2AttributeInfo attrInfo = new SAML2AttributeInfo(
"AttributeWithSingleValue", "ValueOfAttributeWithSingleValue");
attrInfo.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-format:basic");
attrs.add(attrInfo);

ArrayList<String> v = new ArrayList<String>();
v.add("Value1OfAttributeWithMultipleValue");
v.add("Value2OfAttributeWithMultipleValue");
v.add("Value3OfAttributeWithMultipleValue");
SAML2AttributeInfo attrInfo1 = new SAML2AttributeInfo(
"AttributeWithMultipleValue", v);
attrInfo1.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-format:basic");

attrs.add(attrInfo1);

SAML2AttributeInfo attrInfo2 = new SAML2AttributeInfo(
"AttributeWithInvalidNameFormat",
"ValueOfAttributeWithInvalidNameFormatValue");
attrInfo2.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-
format:unspecified");
attrs.add(attrInfo2);

SAML2AttributeInfo attrInfo3 = new SAML2AttributeInfo(
"AttributeWithNullValue", "null");
attrInfo3.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-format:basic");
attrs.add(attrInfo3);
:
:
Collection<SAML2AttributeStatementInfo> attrStatements = new
ArrayList<SAML2AttributeStatementInfo>();
attrStatements.add(new SAML2AttributeStatementInfo(attrs));
attrStatements.add(new SAML2AttributeStatementInfo(attrs1));
return attrStatements;
}
```

How to Implement SAML Attributes

This section walks through the process you follow to implement SAML attributes.

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

> **Note**
>
> This section uses the SAML 2.0 interface names for the purpose of example.

From the SAML credential mapping (Identity Provider) site:

1. Instantiate the `SAML2AttributeInfo` and `SAML2AttributeStatementInfo` classes.

   Implement the `SAML2CredentialAttributeMapper` interface.

   Also implement the `SAML2CredentialNameMapper` interface in the same implementation. (The `SAML2CredentialAttributeMapper` and `SAML2CredentialNameMapper` interfaces must both be in the same implementation.)

   By implementing the `SAML2CredentialNameMapper` interface, you can then use WebLogic Remote Console to set the `NameMapperClassName` attribute to the class name of your `SAML2CredentialAttributeMapper` instance.

2. Use WebLogic Remote Console to configure your new custom attribute mapper on a SAML provider, or on each individual partner, using the `NameMapperClassName` attribute of the SAML Credential Mapping provider to identify it. See Make the Custom SAML Credential Attribute Mapper Class Available in the Console.

3. The SAML Credential Mapping provider determines if the configured custom name mapper is an implementation of the attribute mapping interface and, if so, calls your custom attribute mapping interface to obtain attribute values to write to the generated SAML assertions.

4. The SAML Credential Mapping provider does not validate the attribute names or values obtained from your custom attribute mapper.

   Any attribute with a non-null attribute name is written to the attribute statements in the SAML assertion. An attribute with a null or empty attribute name is ignored, and subsequent attributes are processed.

   If an attribute has multiple values, each value appears as an `<AttributeValue>` element of a single `<Attribute>` in SAML attribute statements.

   For SAML 2.0, null or empty attribute values are handled based on Assertions and the *Protocols for the OASIS SAML V2.0 March 2005* (`http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`).

   An attribute with a name format other than `urn:oasis:names:tc:SAML:2.0:attrname-format:basic` is skipped.

From the SAML Identity Assertion (Service Provider) site:

1. Implement the `SAML2IdentityAsserterAttributeMapper` and `SAML2IdentityAsserterNameMapper` interfaces in the same implementation. (The `SAML2IdentityAsserterAttributeMapper` and `SAML2IdentityAsserterNameMapper` interfaces must both be in the same implementation.)

   By implementing the `SAML2IdentityAsserterNameMapper` interface, you can then use WebLogic Remote Console to set the `NameMapperClassName` attribute to the class name of your `SAML2IdentityAsserterAttributeMapper` instance.

2. Use WebLogic Remote Console to configure the SAML Identity Assertion provider, as described in Make the Custom SAML Identity Asserter Class Available in the Console. Set the `NameMapperClassName` attribute to the class name of your custom `SAML2IdentityAsserterAttributeMapper` instance.

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

The SAML Identity Assertion provider processes `<AttributeStatement>` elements of the incoming SAML assertions and constructs a collection of SAML attribute statements.

3. The SAML Identity Assertion provider determines if the configured custom name mapper implements the `SAML2IdentityAsserterAttributeMapper` interface. If it does, the SAML Identity Assertion provider calls the `mapAttributeInfo` method to obtain the SAML assertion's attributes.

   Your `mapAttributeInfo` method takes a `Collection` of `SAMLAttributeStatementInfo` instances that represent the attributes of attribute statements in a SAML Assertion, and maps the desired attributes in any application specific way.

4. The SAML IdentityAssertion provider makes the attributes from a SAML assertion available to consumers via the Java Subject. This requires that the SAML Authentication provider be configured and the virtual user be enabled on a SAML partner.

   The attributes returned by the mapper are stored as subject principals or private credentials, depending on the class type of the mapped attributes. Specifically, if the mapper returns a collection of `Principal` objects, the mapped attributes are stored into the subject principal set. Otherwise, the subject private credential set is used to carry the mapped attributes.

   The consuming code needs to know the class type of the object that the mapper uses to represent attributes added to the subject, as shown in Example 8-2.

5. The SAML Identity Assertion provider checks the `ContextHandler` and attribute mapper. (This walk through assumes the presence of the attribute mapper as stated in Step 4).

> **Note**
>
> If both the `ContextHandler` and attribute mapper are present and configured, the attributes are instead made available to Web services via the `ContextHandler`, as described in *Securing WebLogic Web Services for Oracle WebLogic Server*.

Examples of the SAML 2.0 Attribute Interfaces

This section provides examples of implementing the SAML 2.0 attribute interfaces that allow writing additional attributes into SAML assertions.

Example Custom SAML 2.0 Credential Attribute Mapper

Example 8-1 shows an example of a single class that implements both the `SAML2CredentialNameMapper` interface and the `SAML2CredentialAttributeMapper` interface.

Example 8-1 SAML 2.0 Credential Attribute Mapper

```
public class CustomSAML2CredentialAttributeMapperImpl implements
SAML2CredentialNameMapper, SAML2CredentialAttributeMapper {
private String nameQualifier = null;

public Collection<SAML2AttributeStatementInfo> mapAttributes(
Subject subject, ContextHandler handler) {
return getAttributeStatementInfo(subject, handler);
}
       /**
         * same as SAML2NameMapperImpl
         */
        public SAML2NameMapperInfo mapName(String name, ContextHandler handler) {
```


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

```
                System.out
                                .println("CustomSAML2CredentialAttributeMapperImpl:mapName
: Mapped name: qualifier: "
                                        + nameQualifier + ", name: " + name);
                return new SAML2NameMapperInfo(nameQualifier, name, null);
        }

        /**
         * same as SAML2NameMapperImpl
         */
        public synchronized void setNameQualifier(String nameQualifier) {
                this.nameQualifier = nameQualifier;
        }

        /**
         * same as SAML2NameMapperImpl
         */
         public SAML2NameMapperInfo mapSubject(Subject subject,
                        ContextHandler handler) {

                // Provider checks for null Subject...
                Set subjects = subject.getPrincipals(WLSUser.class);
                Set groups = subject.getPrincipals(WLSGroup.class);
                String userName = null;
                Set<String> groupStrings = new java.util.HashSet();

                if (subjects == null || subjects.size() == 0) {
                        System.out
                                        .println("CustomSAML2CredentialAttributeMapperImp
l:mapSubject: No valid WLSUser pricipals found in Subject, returning null");
                        return null;
                }

                if (groups == null || groups.size() == 0) {
                        System.out
                                        .println("CustomSAML2CredentialAttributeMapperImp
l:mapSubject: No valid WLSGroup pricipals found in Subject, continuing");
                }
                 else{
                 java.util.Iterator<WLSGroup> it = groups.iterator();
                 while(it.hasNext()){
                 WLSGroup wg = it.next();
                 groupStrings.add(wg.getName());
                 }
                 }

                if (subjects.size() != 1) {
                        System.out
                                        .println("CustomSAML2CredentialAttributeMapperImp
l:mapSubject: More than one WLSUser principal found in Subject, taking first user only");
                }

                userName = ((WLSUser) subjects.iterator().next()).getName();
                if (userName == null || userName.equals("")) {
                        System.out
                                        .println("CustomSAML2CredentialAttributeMapperImp
l:mapSubject: Username string is null or empty, returning null");
                        return null;
                }

                // Return mapping information...
                System.out
```


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

```
TRACE.info("CustomSAML2CredentialAttributeMapperImpl:mapSubject: Mapped subject:
qualifier: "
                                                + nameQualifier
                                                + ", name: "
                                                + userName
                                                + ", groups: "
                                                + groups);

  SAML2NameMapperInfo saml2NameMapperInfo = new
  SAML2NameMapperInfo(nameQualifier, userName, groupStrings);
   //SAML2NameMapperInfo saml2NameMapperInfo = new
  SAML2NameMapperInfo(nameQualifier, userName, groups);

                return new SAML2NameMapperInfo(nameQualifier, userName, groups);
        }

private Collection<SAML2AttributeStatementInfo> getAttributeStatementInfo(
Subject subject, ContextHandler handlers) {
Collection<SAML2AttributeInfo> attrs = new ArrayList<SAML2AttributeInfo>();

SAML2AttributeInfo attrInfo = new SAML2AttributeInfo(
"AttributeWithSingleValue", "ValueOfAttributeWithSingleValue");
attrInfo.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-format:basic");
attrs.add(attrInfo);

ArrayList<String> v = new ArrayList<String>();
v.add("Value1OfAttributeWithMultipleValue");
v.add("Value2OfAttributeWithMultipleValue");
v.add("Value3OfAttributeWithMultipleValue");
SAML2AttributeInfo attrInfo1 = new SAML2AttributeInfo(
"AttributeWithMultipleValue", v);
attrInfo.setAttributeNameFormat("urn:oasis:names:tc:SAML:2.0:attrname-format:basic");

attrs.add(attrInfo1);
:
:
Collection<SAML2AttributeStatementInfo> attrStatements = new
ArrayList<SAML2AttributeStatementInfo>();
attrStatements.add(new SAML2AttributeStatementInfo(attrs));
attrStatements.add(new SAML2AttributeStatementInfo(attrs1));
return attrStatements;
}
}
```

Use WebLogic Remote Console to configure the User Name Mapper class name to the fully-qualified class name of this mapper implementation, as described in Make the Custom SAML Credential Attribute Mapper Class Available in the Console.

The attributes encapsulated in the collection of `SAML2AttributeStatementInfo` objects returned by the custom mapper implementation are included in the generated assertions by the SAML 2.0 Credential Mapping provider.

Custom SAML 2.0 Identity Asserter Attribute Mapper

Example 8-2 shows an example implementation of `SAML2IdentityAsserterNameMapper` and `SAML2IdentityAsserterAttributeMapper`.

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

Example 8-2 Custom SAML 2.0 Identity Asserter Attribute Mapper

```
public class CustomSAML2IdentityAsserterAttributeMapperImpl implements
SAML2IdentityAsserterNameMapper, SAML2IdentityAsserterAttributeMapper {
/**
 * same as SAML2NameMapperImpl
 */
public String mapNameInfo(SAML2NameMapperInfo info, ContextHandler handler) {
// Get the user name ...
String userName = info.getName();
System.out
.println("CustomSAML2IdentityAsserterAttributeMapperImpl:mapNameInfo: returning name: "
+ userName);
return userName;
}
}
:
:
public Collection<Object> mapAttributeInfo0(
Collection<SAML2AttributeStatementInfo> attrStmtInfos,
ContextHandler contextHandler) {
if (attrStmtInfos == null || attrStmtInfos.size() == 0) {
System.out
.println("CustomIAAttributeMapperImpl: attrStmtInfos has no elements");
return null;
}

Collection<Object> customAttrs = new ArrayList<Object>();

for (SAML2AttributeStatementInfo stmtInfo : attrStmtInfos) {
Collection<SAML2AttributeInfo> attrs = stmtInfo.getAttributeInfo();
if (attrs == null || attrs.size() == 0) {
System.out
.println("CustomIAAttributeMapperImpl: no attribute in statement: "
+ stmtInfo.toString());
} else {
for (SAML2AttributeInfo attr : attrs) {
if (attr.getAttributeName().equals("AttributeWithSingleValue")){
    CustomPrincipal customAttr1 = new CustomPrincipal(attr
.getAttributeName(), attr.getAttributeNameFormat(),
attr.getAttributeValues());
    customAttrs.add(customAttr1);
}else{
String customAttr = new StringBuffer().append(
attr.getAttributeName()).append(",").append(
attr.getAttributeValues()).toString();
customAttrs.add(customAttr);
}
}
}
}
return customAttrs;
}

public Collection<Principal> mapAttributeInfo(
Collection<SAML2AttributeStatementInfo> attrStmtInfos,
ContextHandler contextHandler) {
if (attrStmtInfos == null || attrStmtInfos.size() == 0) {
System.out
.println("CustomIAAttributeMapperImpl: attrStmtInfos has no elements");
return null;
}
```


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

```
Collection<Principal> pals = new ArrayList<Principal>();

for (SAML2AttributeStatementInfo stmtInfo : attrStmtInfos) {
Collection<SAML2AttributeInfo> attrs = stmtInfo.getAttributeInfo();
if (attrs == null || attrs.size() == 0) {
System.out
.println("CustomIAAttributeMapperImpl: no attribute in statement: "
+ stmtInfo.toString());
} else {
for (SAML2AttributeInfo attr : attrs) {
CustomPrincipal pal = new CustomPrincipal(attr
.getAttributeName(), attr.getAttributeNameFormat(),
attr.getAttributeValues());
pals.add(pal);
}
}
}
return pals;
}
```

The SAML 2.0 IdentityAssertion provider makes the attributes from a SAML assertion available to consumers via the subject.

Use WebLogic Remote Console to configure the User Name Mapper class name to the fully-qualified class name of this mapper implementation, as described in Make the Custom SAML Identity Asserter Class Available in the Console.

If you are allowing virtual users to log in via SAML, you need to create and configure an instance of the SAML Authentication provider. See Configuring the SAML Authentication Provider.

If the virtual user is enabled and SAML Authenticator provider configured, the attributes returned by the custom attribute mapper are added into the subject.

The attributes returned by the mapper are stored as subject principals or private credentials, depending on the class type of the mapped attributes. Specifically, if the mapper returns a collection of `Principal` objects, the mapped attributes are stored into the subject principal set. Otherwise, the subject private credential set is used to carry the mapped attributes. The example code shows both approaches.

Your application code needs to know the class type of the object that the mapper uses to represent attributes added to the subject. Applications can retrieve the SAML attributes from the subject private credential or principal set, given the class type that the customer attribute mapper uses to represent the attributes.

Make the Custom SAML Credential Attribute Mapper Class Available in the Console

To have the SAML Credential Mapping provider use your `SAML2CredentialAttributeMapper` (SAML 2.0) instance, use WebLogic Remote Console to set the existing `NameMapperClassName` attribute to the class name of this `SAML2CredentialAttributeMapper` instance.

That is, you use the WebLogic Remote Console field for the name mapper class name attribute to specify the class name of the `SAML2CredentialAttributeMapper` instance in the active security realm.

To use a custom user name mapper with the WebLogic SAML Credential Mapping provider

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

1. In WebLogic Remote Console, go to the **Edit Tree**, then **Security**, then **Realms**, then *myRealm*, then **Credential Mappers**, select the name of a SAML Credential Mapping Version 2 provider.
2. On the **SAML Credential Mapper V2 Parameters** tab, in the **Name Mapper Class Name** field, enter the class name of your `SAML2CredentialAttributeMapper` implementation. The class name must be in the system classpath.
3. Click **Save**.

Make the Custom SAML Identity Asserter Class Available in the Console

To have the SAML Identity Assertion provider use this `SAML2IdentityAsserterAttributeMapper` (SAML 2.0) instance, you can use WebLogic Remote Console to set the existing `NameMapperClassName` attribute to the class name of this `SAML2IdentityAsserterAttributeMapper` instance.

That is, you use the Console control for the name mapper class name attribute to specify the class name of the `SAML2IdentityAsserterAttributeMapper` instance in the active security realm.

To use a custom user name mapper with the WebLogic SAML Identity Asserter provider:

1. In WebLogic Remote Console, go to the **Edit Tree**, then **Security**, then **Realms**, then *myRealm*, then **Authentication Providers**, select the name of a SAML Identity Asserter Version 2 provider.
2. On the **SAML Identity Asserter V2 Parameters** tab, in the **Name Mapper Class Name** field, enter the class name of your `SAML2IdentityAsserterAttributeMapper` implementation. The class name must be in the system classpath.
3. Click **Save**.

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

9

Using CertPath Building and Validation

The WebLogic Security service provides the Certificate Lookup and Validation (CLV) API that finds and validates X509 certificate chains. Use the CertPath providers provided by Oracle WebLogic Server to build and validate certificate chains, or any custom CertPath providers. A CertPath is a JDK class that stores a certificate chain in memory. The term CertPath is also used to refer to the JDK architecture and framework that is used to locate and validate certificate chains. The CLV framework extends and completes the JDK CertPath functionality. CertPath providers rely on a tightly coupled integration of WebLogic and JDK interfaces.

This chapter includes the following sections:

- CertPath Building
- CertPath Validation
- Instantiate a CertPathSelector
- Instantiate a CertPathBuilderParameters
- Use the JDK CertPathBuilder Interface
- Instantiate a CertPathValidatorParameters
- Use the JDK CertPathValidator Interface

CertPath Building

To use a CertPath Builder in your application, you must perform a sequence of steps such as, instantiating a `CertPathSelector` object, instantiating a `CertPathBuilderParameters` object, and implementing the JDK `CertPathBuilder` interface.

1. Instantiate a CertPathSelector
2. Instantiate a CertPathBuilderParameters
3. Use the JDK CertPathBuilder Interface

Instantiate a CertPathSelector

The `CertPathSelector` interface (`weblogic.security.pk.CertPathSelector`) contains the selection criteria for locating and validating a certification path. Because there are many ways to look up certification paths, a derived class is created for each type of selection criteria.

Each selector class has one or more methods to retrieve the selection data and a constructor.

The classes in `weblogic.security.pk` that implement the `CertPathSelector` interface, one for each supported type of certificate chain lookup, are as follows:

- EndCertificateSelector – used to find and validate a certificate chain given its end certificate.
- IssuerDNSerialNumberSelector – used to find and validate a certificate chain from its end certificate's issuer DN and serial number.

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

- SubjectDNSelector – used to find and validate a certificate chain from its end certificate's subject DN.
- SubjectKeyIdentifierSelector – used to find and validate a certificate chain from its end certificate's subject key identifier (an optional field in X509 certificates).

> **Note**
>
> The selectors that are supported depend on the configured CertPath providers. The configured CertPath providers are determined by the administrator.
>
> The WebLogic CertPath provider uses only the EndCertificateSelector selector.

Example 9-1 shows an example of choosing a selector.

Example 9-1 Make a certificate chain selector

```
// you already have the end certificate
// and want to use it to lookup and
// validate the corresponding chain
X509Certificate endCertificate = ...
// make a cert chain selector
CertPathSelector selector = new EndCertificateSelector(endCertificate);
```

Instantiate a CertPathBuilderParameters

You pass an instance of `CertPathBuilderParameters` as the `CertPathParameters` object to the JDK's `CertPathBuilder.build()` method.

The following constructor and method are provided:

- CertPathBuilderParameters

  ```
  public CertPathBuilderParameters(String realmName,
                                   CertPathSelector selector,
                                   X509Certificate[]
                                   trustedCAs,
                                   ContextHandler context)
  ```

  Constructs a `CertPathBuilderParameters` object.

  You must provide the realm name. To do this, get the domain's SecurityConfigurationMBean. Then, get the SecurityConfigurationMBean's default realm attribute, which is a realm MBean. Finally, get the realm MBean's name attribute. You must use the runtime JMX MBean server to get the realm name.

  You must provide the `selector`. You use one of the `weblogic.security.pk.CertPathSelector` interfaces derived classes, described in Instantiate a CertPathSelector to specify the selection criteria for locating and validating a certification path.

  Specify trusted CAs if you have them. Otherwise, the server's trusted CAs are used. These are just a hint to the configured CertPath builder and CertPath validators which, depending on their lookup/validation algorithm, may or may not use these trusted CAs.

  ContextHandler is used to pass in an optional list of name/value pairs that the configured `CertPathBuilder` and `CertPathValidators` may use to look up and validate the chain. It is symmetrical with the context handler passed to other types of security providers. Setting context to null indicates that there are no context parameters.

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

- clone

  ```
  Object clone()
  ```

  This interface is not cloneable.

  Example 9-2 shows an example of passing an instance of `CertPathBuilderParameters`.

Example 9-2 Pass An Instance of CertPathBuilderParameters

```
// make a cert chain selector
CertPathSelector selector = new EndCertificateSelector(endCertificate);
String realm = _;
// create and populate a context handler if desired, or null
ContextHandler context = _;
// pass in a list of trusted CAs if desired, or null
X509Certificate[] trustedCAs = _;
// make the params
CertPathBuilderParams params =
new CertPathBuilderParameters(realm, selector, context, trustedCAs);
```

Use the JDK CertPathBuilder Interface

The `java.security.cert.CertPathBuilder` class is the base class for creating the `CertPathBuilder` object. To use the JDK `CertPathBuilder` interface, do the following:

1. Call the static `CertPathBuilder.getInstance` method to retrieve the CLV framework's CertPathBuilder. You must specify `WLSCertPathBuilder` as the algorithm name that's passed to the call.
2. Once the `CertPathBuilder` object has been obtained, call the "build" method on the returned CertPathBuilder. This method takes one argument - a `CertPathParameters` that indicates which chain to find and how it should be validated.

   You must pass an instance of `weblogic.security.pk.CertPathBuilderParameters` as the `CertPathParameters` object to the JDK's CertPathBuilder.build() method, as described in Instantiate a CertPathBuilderParameters.
3. If successful, the result (including the CertPath that was built) is returned in an object that implements the `CertPathBuilderResult` interface. The builder determines how much of the CertPath is returned.
4. If not successful, the CertPathBuilder build method throws InvalidAlgorithmParameterException if the params is not a WebLogic `CertPathBuilderParameters`, if the configured CertPathBuilder does not support the selector, or if the realm name does not match the realm name of the default realm from when the server was booted.

   The `CertPathBuilder` build method throws `CertPathBuilderException` if the cert path could not be located or if the located cert path is not valid

Example Code Flow for Looking Up a Certificate Chain

Example 9-3 Looking up a Certificate Chain

```
import weblogic.security.pk.CertPathBuilderParameters;
import weblogic.security.pk.CertPathSelector;
import weblogic.security.pk.EndCertificateSelector;
import weblogic.security.service.ContextHandler;
import java.security.cert.CertPath;
import java.security.cert.CertPathBuilder;
import java.security.cert.X509Certificate;
```


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

```
// you already have the end certificate
// and want to use it to lookup and
// validate the corresponding chain
X509Certificate endCertificate = ...

// make a cert chain selector
CertPathSelector selector = new EndCertificateSelector(endCertificate);

String realm = _;

// create and populate a context handler if desired
ContextHandler context = _;

// pass in a list of trusted CAs if desired
X509Certificate[] trustedCAs = _;

// make the params
CertPathBuilderParams params =
new CertPathBuilderParameters(realm, selector, context, trustedCAs);
// get the WLS CertPathBuilder
CertPathBuilder builder =
CertPathBuilder.getInstance("WLSCertPathBuilder");

// use it to look up and validate the chain
CertPath certpath = builder.build(params).getCertPath();
X509Certificate[] chain =
certpath.getCertificates().toArray(new X509Certificate[0]);
```

CertPath Validation

To use a CertPath Validator in your application, you must instantiate a `CertPathValidatorParameters` and use the JDK `CertPathValidator` interface.

1. Instantiate a CertPathValidatorParameters
2. Use the JDK CertPathValidator Interface

Instantiate a CertPathValidatorParameters

You pass an instance of `CertPathValidatorParameters` as the `CertPathParameters` object to the JDK's `CertPathValidator.validate()` method.

The following constructor and method are provided:

- CertPathValidatorParameters

  ```
  public CertPathValidatorParameters(String realmName,
                                     X509Certificate[] trustedCAs,
                                     ContextHandler context)
  ```

  Constructs a CertPathValidatorParameters.

  You must provide the realm name. To do this, get the domain's `SecurityConfigurationMBean`. Then, get the default realm attribute of the `SecurityConfigurationMBean`, which is a realm MBean. Finally, get the realm MBean's name attribute. You must use the runtime JMX MBean server to get the realm name.

  Specify trusted CAs if you have them. Otherwise, the server's trusted CAs are used. These are just a hint to the configured CertPath builder and CertPath validators which, depending on their lookup/validation algorithm, may or may not use these trusted CAs.

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

ContextHandler is used to pass in an optional list of name/value pairs that the configured CertPathBuilder and CertPathValidators may use to look up and validate the chain. It is symmetrical with the context handler passed to other types of security providers. Setting context to null indicates that there are no context parameters.

- clone

  ```
  Object clone()
  ```

  This interface is not cloneable.

  Example 9-4 shows an example of passing an instance of CertPathValidatorParameters.

Example 9-4 Pass an Instance of CertPathValidatorParameters

```
// get the WLS CertPathValidator
CertPathValidator validator =
CertPathValidator.getInstance("WLSCertPathValidator");

String realm = _;

// create and populate a context handler if desired, or null
ContextHandler context = _;

// pass in a list of trusted CAs if desired, or null
X509Certificate[] trustedCAs = _;

// make the params (for the default security realm)
CertPathValidatorParams params =
new CertPathValidatorParams(realm, context, trustedCAs);
```

Use the JDK CertPathValidator Interface

The `java.security.cert.CertPathValidator` class is the base class for creating a `CertPathValidator` object. To use the JDK `CertPathValidator` interface, do the following:

1. Call the static `CertPathValidator.getInstance` method to retrieve the CLV framework's CertPathValidator. You must specify `WLSCertPathValidator` as the algorithm name that's passed to the call.
2. Once the `CertPathValidator` object has been obtained, call the `validate` method on the returned CertPathValidator. This method takes one argument - a `CertPathParameters` that indicates how it should be validated.

   You must pass an instance of `weblogic.security.pk.CertPathValidatorParameters` as the `CertPathParameters` object to the JDK's `CertPathValidator.validate()` method, as described in Instantiate a CertPathValidatorParameters.
3. If successful, the result is returned in an object that implements the `CertPathValidatorResult` interface.
4. If not successful, the `CertPathValidator.validate()` method throws `InvalidAlgorithmParameterException` if params is not a WebLogic `CertPathValidatorParameters` or if the realm name does not match the realm name of the default realm from when the server was booted.

   The CertPathValidator validate method throws `CertPathValidatorException` if the certificates in the CertPath are not ordered (the end certificate must be the first cert) or if the CertPath is not valid.

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

Example Code Flow for Validating a Certificate Chain

Example 9-5 Performing Extra Validation

```
import weblogic.security.pk.CertPathValidatorParams;
import weblogic.security.service.ContextHandler;
import java.security.cert.CertPath;
import java.security.cert.CertPathValidator;
import java.security.cert.X509Certificate;

// you already have an unvalidated X509 certificate chain
// and you want to get it validated
X509Certificate[] chain = ...

// convert the chain to a CertPath
CertPathFactory factory = CertPathFactory.getInstance("X509");
ArrayList list = new ArrayList(chain.length);
for (int i = 0; i < chain.length; i++) {
list.add(chain[i]);
}
CertPath certPath = factory.generateCertPath(list);

// get the WLS CertPathValidator
CertPathValidator validator =
CertPathValidator.getInstance("WLSCertPathValidator");

String realm = _;

// create and populate a context handler if desired, or null
ContextHandler context = _;

// pass in a list of trusted CAs if desired, or null
X509Certificate[] trustedCAs = _;

// make the params (for the default security realm)
CertPathValidatorParams params =
new CertPathValidatorParams(realm, context, trustedCAs);

// use it to validate the chain
validator.validate(certPath, params);
```


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

10

Using Jakarta Authentication for a Web Application

Oracle WebLogic Server supports the use of Jakarta Authentication to configure an Authentication Configuration Provider for a Web application and using that instead of the default WebLogic Server authentication mechanism for that Web application. Learn how to configure Jakarta Authentication for the deployed web application.
Jakarta Authentication was previously called Java Authentication Service Provider Interface for Containers (JASPIC).

- Overview of Jakarta Authentication
- Do You Need to Implement an Authentication Configuration Provider?
- Do You Need to Implement a Principal Validation Provider?
- Implement a SAM
- Configure Jakarta Authentication for the Deployed Web Application

This section assumes that you are familiar with a basic overview of Jakarta Authentication, as described in Jakarta Authentication Security in *Understanding Security for Oracle WebLogic Server*.

Overview of Jakarta Authentication

The Authentication Configuration provider in Jakarta Authentication assumes responsibility for authenticating the user credentials for a Web application and returning a subject. It authenticates incoming Web application messages and returns the identity (the expected subject) established as a result of the message authentication to WebLogic Server.

The Jakarta Authentication programming model is described in the Jakarta Authentication specification, https://jakarta.ee/specifications/authentication/. It defines a service provider interface (SPI) by which authentication providers that implement message authentication mechanisms can be integrated in server Web application message processing containers or runtimes.

WebLogic Server allows you to use Jakarta Authentication to delegate authentication for Web applications to your configured Authentication Configuration providers. You do not have to modify your Web application code to use Jakarta Authentication. Instead, you use WebLogic Remote Console or WLST to enable Jakarta Authentication for the Web application post deployment.

For each of your deployed Web applications in the domain, determine whether you want Jakarta Authentication to be disabled (the default), or select one of your configured Authentication Configuration providers to authenticate the user credentials and return a valid subject. If you configure an Authentication Configuration provider for a Web application, it is used instead of the WLS authentication mechanism for that Web application. You should therefore exercise care when you specify an Authentication Configuration provider to make sure that it satisfies your security authentication needs.

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

Do You Need to Implement an Authentication Configuration Provider?

If you have a specific requirement that is not addressed by the default WebLogic Authentication provider, then you can implement your own Authentication Configuration provider.

You can use either the default WebLogic Server Authentication Configuration provider, or you can implement your own. To use the default WebLogic Server Authentication Configuration provider and configure it, see the steps described in Configuring Jakarta Authentication Security in *Administering Security for Oracle WebLogic Server*.

As described in the Jakarta Authentication specification, https://jakarta.ee/specifications/authentication/, the Authentication Configuration provider (called "authentication context configuration provider" in the specification) is an implementation of the `jakarta.security.auth.message.config.AuthConfigProvider` interface.

The Authentication Configuration provider provides a configuration mechanism used to define the registered Server Authentication Modules (SAM's) and bindings to applications that require protection from unauthenticated/authorized access.

Do You Need to Implement a Principal Validation Provider?

Authentication providers rely on Principal Validation providers to sign and verify the authenticity of principals (users and groups) contained within a subject. The Principal Validation provider, thus, prevents malicious individuals from tampering with the principals stored in a subject.

Principals are sent to the specified Principal Validation provider, which signs the principals and then returns them to the client application via WebLogic Server. Whenever the principals stored within the subject are required for other security operations, the same Principal Validation provider will verify that the principals stored within the subject have not been modified since they were signed.

Such verification provides an additional level of trust and may reduce the likelihood of malicious principal tampering. The authenticity of the subject's principals is also verified when making authorization decisions.

You must therefore use a Principal Validation provider as described in Principal Validation Providers.

Whether you use the existing WebLogic Principal Validation provider or implement a custom Principal Validation provider depends on the type of principals you are using:

- WebLogic Server principals — The WebLogic Principal Validation provider includes implementations of the `WLSUser` and `WLSGroup` interfaces, named `WLSUserImpl` and `WLSGroupImpl`. These are located in the `weblogic.security.principal` package.

  It also includes an implementation of the `PrincipalValidator` SSPI called `PrincipalValidatorImpl` (located in the `com.bea.common.security.provider` package). To use this class, make the `PrincipalValidatorImpl` class the runtime class for your Principal Validation provider. See the `PrincipalValidator` SSPI for usage information.
- Custom Principals — If you have your own validation scheme and do not want to use the WebLogic Principal Validation provider, or if you want to provide validation for principals other than WebLogic Server principals, then you need to develop a custom Principal Validation provider.

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

> **Note**
>
> If you add custom principals, you must add a Principal Validation provider or authorization fails. The WebLogic Server security framework performs principal validation as part of authorization. (The only exception is if you are using Jakarta Authorization for authorization. Even in the case of Jakarta Authorization, if your Web application or EJB accesses any other server resource (for example, JDBC), WebLogic Server authorization and principal validation are used.)

In this case, you must also develop an Authentication provider. The `AuthenticationProviderV2` SSPI includes a method called `getPrincipalValidator` in which you specify the Principal Validation provider's runtime class. WebLogic Server uses this method to get the Principal Validation provider. (In this use, the other methods can return null.)

Both options are described in Principal Validation Providers in *Developing Security Providers for Oracle WebLogic Server*.

Implement a SAM

A key step in adding an authentication mechanism to a compatible server-side message processing runtime is acquiring a Server Authentication Module (SAM) that implements the desired authentication mechanism.

You must implement your own SAM that works with the default WebLogic Server Authentication Configuration provider, or with your own Authentication Configuration provider.

The SAM represents the implementation of a server-side authentication provider that is compliant with Jakarta Authentication. As described in the Jakarta Authentication specification https://jakarta.ee/specifications/authentication/, a SAM implements the `jakarta.security.auth.message.module.ServerAuthModule` interface and is invoked by WebLogic Server at predetermined points in the message processing model.

> **Note**
>
> A sample SAM implementation is described in Adding Authentication Mechanisms to the Servlet Container in the *GlassFish Server Open Source Edition Application Development Guide*. Although written from the GlassFish Server perspective, the tips for writing a SAM, and the sample SAM itself, are instructive.

Configure Jakarta Authentication for the Deployed Web Application

To configure Jakarta Authentication for your deployed Web application, you must add the jar for your SAM to the system classpath using the command line, enable Jakarta Authentication in your domain using WebLogic Remote Console, and configure the desired Authentication Configuration provider to specify the classname of the SAM.

Perform the following steps to configure Jakarta Authentication for a Web application:

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

1. Add the jar for your SAM to the system classpath via the startup scripts or the command line used to start the WebLogic Server instance.

   If you also configured a custom Authentication Configuration provider, you must add the jar for your custom Authentication Configuration provider to the system classpath via the startup scripts or the command line used to start the WebLogic Server instance.

2. Enable Jakarta Authentication in the domain, as described in Configuring Jakarta Authentication Security in *Administering Security for Oracle WebLogic Server*.

3. Configure the WebLogic Server Authentication Configuration provider or the custom Authentication Configuration provider to specify the classname of the SAM as described in Configuring Jakarta Authentication Security in *Administering Security for Oracle WebLogic Server*.

4. Configure Jakarta Authentication for the application as described in Configure JASPIC for a Web Application in *Oracle WebLogic Remote Console Online Help*.

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

11

Using Jakarta Security

The Jakarta Security specification defines portable, plug-in interfaces for HTTP authentication and identity stores, and an injectable `SecurityContext` interface that provides an API for programmatic security. You can use the built-in implementations of the plug-in SPIs, or write custom implementations.

Using Jakarta Security, you can define all of the security information directly within the application. Bundling the security configuration in the application instead of configuring it externally improves the management of the application's lifecycle, especially in a world of Docker-hosted microservices that are distributed in containers.

- Overview of Jakarta Security in WebLogic Server
- About the HttpAuthenticationMechanism Interface
- About the Identity Store Interfaces
- Usage Requirements

Overview of Jakarta Security in WebLogic Server

Oracle WebLogic Server supports the Jakarta Security specification which defines portable authentication mechanisms (such as `HttpAuthenticationMechanism` and `IdentityStore`), and an access point for programmatic security using the `SecurityContext` interface. In WebLogic Server, these authentication mechanisms are supported in the web container, and the `SecurityContext` interfaces are supported in the Servlet and EJB containers.

The programming model for Jakarta Security is defined in the specification at `https://jakarta.ee/specifications/security/`. WebLogic Server supports the plug-in interface for authentication, `HttpAuthenticationMechanism`, and includes built-in support for the BASIC, FORM, and Custom FORM authentication mechanisms defined in the specification. WebLogic Server also supports the `RememberMeIdentityStore` interface, and built-in implementations of the `IdentityStore` interface (LDAP identity store and Database identity store) as well as the custom identity store.

The `SecurityContext` interfaces for web applications and EJBs are described in Authenticating Users Programmatically and Using Programmatic Security With EJBs, respectively.

The `HttpAuthenticationMechanism` interface is designed to capitalize on the strengths of existing Servlet and authentication mechanisms from Jakarta Authentication. An `HttpAuthenticationMechanism` is a CDI bean, and is therefore made available to the container automatically by CDI (see Using Contexts and Dependency Injection for the Jakarta EE Platform in *Developing Applications for Oracle WebLogic Server* for more information on CDI support). The container is responsible for placing the `HttpAuthenticationMechanism` into service. The `IdentityStore` interface is intended primarily for use by `HttpAuthenticationMechanism` implementations, but could in theory be used by other types of authentication mechanisms (such as a Jakarta Authentication `ServerAuthModule`). `HttpAuthenticationMechanism` implementations are not required to use `IdentityStore` — they can authenticate users in any manner they choose — but the `IdentityStore` interface is a useful and convenient mechanism.

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

A significant advantage of using the `HttpAuthenticationMechanism` and `IdentityStore` interfaces over the declarative mechanisms defined by the Servlet specification is that they allow an application to control the identity stores that it authenticates against in a standard, portable way. Because implementations of these SPI interfaces are CDI beans, applications can provide implementations that support application-specific authentication mechanisms, or validate user credentials against application-specific identity stores, simply by including them in a bean archive that is part of the deployed application.

About the HttpAuthenticationMechanism Interface

The `HttpAuthenticationMechanism` interface defines an SPI for writing authentication mechanisms that can be provided with an application and deployed using CDI. Developers can write their own implementations of `HttpAuthenticationMechanism` to support specific authentication token types or protocols. There are also several built-in authentication mechanisms that perform BASIC, FORM, and Custom FORM authentication.

The built-in authentication mechanisms are enabled and configured using annotations that, when used, make the corresponding built-in mechanism available as a CDI bean. The Jakarta Security specification also supports the use of Expression Language 4.0 in these annotations to allow dynamic configuration. For more information about Java Expression Language (EL), see the Expression Language specification at https://jakarta.ee/specifications/expression-language/.The annotations for the built-in authentication mechanisms are as follows:

- `BasicAuthenticationMechanismDefinition` — implements BASIC authentication that conforms to the behavior of the servlet container when `BASIC <auth-method>` is declared in `web.xml`. In BASIC authentication, the web client obtains the user name and the password from the user and transmits them to the web server. The web server then authenticates the user in the specified realm.
- `FormAuthenticationMechanismDefinition` — implements FORM authentication that conforms to the behavior of the servlet container when the `FORM <auth-method>` is declared in `web.xml`. FORM Based Authentication introduces a required form-based authentication mechanism that allows a developer to control the look and feel of the login screens. The web application deployment descriptor contains entries for a login form and error page. The login form must contain fields for entering a user name and password.
- `CustomFormAuthenticationMechanismDefinition` — implements a modified version of FORM authentication. In WebLogic Server, the difference is that authentication occurs by invoking `SecurityContext.authenticate()` using the credentials the application collected.

An implementation of `HttpAuthenticationMechanism` must be a CDI bean to be recognized and deployed at runtime, and is assumed to be application scoped. During bean discovery, the servlet container looks for a bean that implements `HttpAuthenticationMechanism` — there should be only one per application — and, if found, arranges for it to be deployed to authenticate the application's callers.

The servlet container leverages Jakarta Authentication, to deploy authentication mechanisms. The container provides a Jakarta Authentication Server Auth Module (SAM) that can delegate to an `HttpAuthenticationMechanism`, and arranges for that "bridge" SAM to be registered with the Jakarta Authentication `AuthConfigFactory`. At runtime, normal Jakarta Authentication processing invokes the bridge SAM, which then delegates to the `HttpAuthenticationMechanism` to perform the authentication and drive any necessary dialog with the caller, or with third parties involved in the authentication protocol flow.

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

HttpAuthenticationMechanism Interface Methods

The `HttpAuthenticationMechanism` interface defines three interface methods, which correspond to the three methods defined by the Jakarta Authentication ServerAuth interface.

When one of the Jakarta Authentication methods is invoked on the bridge SAM, it delegates to the corresponding method of the `HttpAuthenticationMechanism`. Although the method names are identical, the method signatures are not; the bridge SAM maps back and forth between the parameters passed to it by the Jakarta Authentication framework, and the parameters expected by an `HttpAuthenticationMechanism`.

The three `HttpAuthenticationMechanism` interface methods are as follows:

- `validateRequest()` — validate an incoming request and authenticate the caller.
- `secureResponse()` — secure a response message. This method is optional if the default is sufficient.
- `cleanSubject()` — clear the provided Subject of principals and credentials. This method is optional if the default is sufficient.

Only the `validateRequest()` method must be implemented by an `HttpAuthenticationMechanism`; the interface includes default implementations for `secureResponse()` and `cleanSubject()` that will often be sufficient.

HttpAuthenticationMechanism Interface Annotations

You can use the following annotations to add additional behaviors to an `HttpAuthenticationMechanism`:

- `AutoApplySession` — provides an application with a way to declaratively enable Jakarta Authentication `jakarta.servlet.http.registerSession` behavior for an authentication mechanism, and automatically apply it for every request.
- `LoginToContinue` — provides an application with the ability to declaratively add "login to continue" functionality to an authentication mechanism. The annotation is also used to configure the login page, error page, and redirect/forward behavior for the built-in form-based authentication mechanisms.
- `RememberMe` - specifies that a `RememberMe` identity store should be used to enable `RememberMe` functionality for the authentication mechanism. To use `RememberMe`, the application must provide its implementation of HAM and annotate the HAM with the `RememberMe` annotation.

About the Identity Store Interfaces

In WebLogic Server, all built-in authentication mechanisms need to be authenticated using an identity store. The Jakarta Security specification defines two identity store interfaces, `IdentityStore` and `RememberMeIdentityStore`. The `IdentityStore` interface defines methods for validating a caller's credentials, such as username and password, and returning group membership information. The `RememberMeIdentityStore` interface is a variation on the `IdentityStore` interface intended specifically to address cases where the identity of an authenticated user should be remembered for an extended period of time.

The following topics describe the identity store interfaces in more detail:

- IdentityStore Interface

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

- RememberMeIdentityStore Interface

IdentityStore Interface

The `IdentityStore` interface defines an SPI for interacting with identity stores, which are directories or databases containing user account information. An implementation of the `IdentityStore` interface can validate users' credentials, provide information about the groups they belong to, or both. Most often, an `IdentityStore` implementation will interact with an external identity store — an LDAP server, for example — to perform the actual credential validation and group lookups, but an `IdentityStore` may also manage user account data itself.

There are two built-in implementations of `IdentityStore`: an LDAP identity store, and a Database identity store. These identity stores delegate to external stores that must already exist; the `IdentityStore` implementations do not provide or manage the external store. Use the following annotations to configure communication between the `IdentityStore` interface and an external store:

- `LdapIdentityStoreDefinition` — configures an identity store with the parameters necessary to communicate with an external LDAP server, validate user credentials, and/or lookup user groups.
- `DatabaseIdentityStoreDefinition` — configures an identity store with the parameters necessary to connect to an external database, validate user credentials, and/or lookup user groups. You must supply a `PasswordHash` implementation when configuring a Database Identity Store.

An application can provide its own custom identity store, or use the built-in LDAP or database identity stores. WebLogic Server provides an optional example demonstrating the use of a built-in database identity store. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

An implementation of `IdentityStore` must be a CDI bean to be recognized and deployed at runtime, and is assumed to be application scoped. Multiple implementations of `IdentityStore` may be present. If so, they are invoked under the control of an `IdentityStoreHandler`.

IdentityStoreHandler

Authentication mechanisms do not interact with `IdentityStore` directly; instead, they call an `IdentityStoreHandler`. An implementation of the `IdentityStoreHandler` interface provides a single method, `validate(Credential)`, which, when invoked, iterates over the available IdentityStores and returns an aggregated result. An `IdentityStoreHandler` must also be a CDI bean, and is assumed to be application scoped. At runtime, an authentication mechanism injects the `IdentityStoreHandler` and invokes on it. The `IdentityStoreHandler`, in turn, looks up the available IdentityStores and invokes on them to determine the aggregate result.

There is a built-in `IdentityStoreHandler` that implements a standard algorithm defined by the Jakarta Security specification. An application may also supply its own `IdentityStoreHandler`, which can use any desired algorithm to select and invoke on IdentityStores, and return an aggregated (or non-aggregated) result.

IdentityStore Interface Methods

The `IdentityStore` interface has four methods:

- `validate(Credential)` — validate a Credential, and return the result of that validation.

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

- `getCallerGroups(CredentialValidationResult)` — return the groups associated with the caller indicated by the supplied `CredentialValidationResult`, which represents the result of a previous, successful validation.
- `validationTypes()` — returns a Set of validation types (one or more of `VALIDATE`, `PROVIDE_GROUPS`) that indicate the operations supported by this instance of the `IdentityStore`.
- `priority()` — returns a positive integer representing the self-declared priority of this `IdentityStore`. Lower values represent higher priority.

Because `getCallerGroups()` is a sensitive operation — it can return information about arbitrary users, and does not require that the caller provide the user's credential or proof of identity — the caller should have the `IdentityStorePermission("getGroups")` permission. For this permission check to be performed, ensure that the Java Security Manager is enabled. See Using the Java Security Manager to Protect WebLogic Resources.

RememberMeIdentityStore Interface

The `RememberMeIdentityStore` interface represents a special type of identity store. It is not directly related to the `IdentityStore` interface; that is, it does not implement or extend it. It does, however, perform a similar, albeit specialized, function. You use the `RememberMeIdentityStore` interface when an application wants to "remember" a user's authenticated session for an extended period, so that the caller can return to the application periodically without needing to present primary authentication credentials each time. For example, a web site may remember you when you visit, and prompt for your password only periodically, perhaps once every two weeks, as long as you don't explicitly log out.

RememberMe works as follows:

- When a request from an unauthenicated user is received, the user is authenticated using an `HttpAuthenticationMechanism` that is provided by the application (this is required — `RememberMeIdentityStore` can only be used in conjunction with an application-supplied `HttpAuthenticationMechanism`).
- After authentication, the configured `RememberMeIdentityStore` saves information about the user's authenticated identity, so that it can be restored later, and generates a long-lived "remember me" login token that is sent back to the client, perhaps as a cookie.
- On a subsequent visit to the application, the client presents the login token. The `RememberMeIdentityStore` then validates the token and returns the stored user identity, which is then established as the user's authenticated identity. If the token is invalid or expired, it is discarded, the user is authenticated normally again, and a new login token is generated.

The `RememberMeIdentityStore` interface defines the following methods:

- `generateLoginToken(CallerPrincipal caller, Set<String> groups)` — generate a login token for a newly authenticated user, and associate it with the provided caller/group information.
- `removeLoginToken(String token)` — remove the (presumably expired or invalid) login token and any associated caller/group information.
- `validate(RememberMeCredential credential)` — validate the supplied credential, and, if valid, return the associated caller/group information. (RememberMeCredential is essentially just a holder for a login token).

An implementation of `RememberMeIdentityStore` must be a CDI bean, and is assumed to be application scoped. You configure a `RememberMeIdentityStore` by adding a `RememberMe`

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

annotation to an application's `HttpAuthenticationMechanism`, which indicates that a `RememberMeIdentityStore` is in use, and provides related configuration parameters. A container-supplied interceptor then intercepts calls to the `HttpAuthenticationMechanism`, invokes the `RememberMeIdentityStore` as necessary before and after calls to the authentication mechanism, and ensures that the user's identity is correctly set for the session. See the Jakarta Security specification for a detailed description of the required interceptor behavior.

Implementations of `RememberMeIdentityStore` should take care to manage tokens and user identity information securely. For example, login tokens should not contain sensitive user information, like credentials or sensitive attributes, to avoid exposing that information if an attacker were able to gain access to the token — even an encrypted token is potentially vulnerable to an attacker with sufficient time/resources. Similarly, tokens should be encrypted/signed wherever possible, and sent only over secure channels (HTTPS). User identity information managed by a `RememberMeIdentityStore` should be stored as securely as possible (but does not necessarily need to be reliably persisted — the only impact of a "forgotten" session is that the user will be prompted to log in again).

Usage Requirements

Using the Jakarta Security authentication mechanisms does not require any specific configuration, but you must ensure that other functionality, such as Jakarta Authentication and CDI, is enabled.

To use the Jakarta Security features in WebLogic Server, note the following requirements:

- Web applications must include the `beans.xml` deployment descriptor file in the application's WAR or EAR file, as specified by the CDI specification. Because the `HttpAuthenticationMechanism` and `IdentityStore` interfaces are implemented as CDI beans, they are visible to the container through CDI.
- The `metadata-complete` attribute in the `web.xml` file for the web applications must *NOT* be set to `true`. The default in WebLogic Server is `false`.
- Jakarta Authentication must be enabled at the domain level. By default, Jakarta Authentication is enabled for a domain in WebLogic Server.

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

A

Deprecated Security APIs

Some or all of the Security interfaces, classes, and exceptions in the WebLogic security packages, `weblogic.security.service` and `weblogic.security.SSL`, were deprecated prior to the current release of Oracle WebLogic Server. For specific information on the interfaces, classes, and exceptions deprecated in each package, see the *Java API Reference for Oracle WebLogic Server*.

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