Oracle® Fusion Middleware

Developing JMS Applications for Oracle WebLogic Server



26 (26.1.0.0.0)
G56953-01
November 2026

ORACLE

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

G56953-01

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 i
Diversity and Inclusion i
Related Documentation i
Conventions ii

1 Understanding WebLogic JMS

Overview of Jakarta Messaging and WebLogic JMS 1
- What Is Jakarta Messaging? 1
- Messaging Specification and API Version 2
- WebLogic JMS Architecture 2

Understanding the Messaging Models 3
- Point-to-Point Messaging 4
- Publish/Subscribe Messaging 5
- Message Persistence 5

Understanding the JMS API 6
- ConnectionFactory 7
  - Using the Default Connection Factories 7
  - Configuring and Deploying Connection Factories 8
  - The ConnectionFactory Interface 8
- JMSContext 9
- Connection 9
- Session 10
  - WebLogic JMS Session Guidelines 10
  - Domain-Specific Session Interfaces 10
  - Non-Transacted Sessions 10
  - Transacted Sessions 11
- Destination 12
  - Distributed Destinations 13
- MessageProducer and MessageConsumer 13
- Messages 14
  - Message Header Fields 14

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

Message Property Fields 17
Message Body 17
ServerSessionPoolFactory 18
ServerSessionPool 19
ServerSession 19
ConnectionConsumer 19
Value-Added Public JMS API Extensions 19
WebLogic Server Value-Added JMS Features 20

2 Best Practices for Application Design

Message Design 1
Serializing Application Objects 1
Serializing Strings 1
Server-side Serialization 1
Selection 1
Message Compression 1
Message Properties and Message Header Fields 2
Message Ordering 2
Topics Vs. Queues 2
Asynchronous Vs. Synchronous Consumers 3
Persistent Vs. Non Persistent Messages 3
Deferring Acknowledges and Commits 4
Acknowledgement Optimization for Nondurable Subscribers 5
Alternative Qualities of Service, Multicast and No-Acknowledge 5
Using MULTICAST_NO_ACKNOWLEDGE 5
Using NO_ACKNOWLEDGE 5
Use Sessions from One Thread at a Time 6
Using the JMSXUserID Property 6
Performance and Tuning 7

3 Enhanced Support for Using WebLogic JMS with EJBs and Servlets

Enabling WebLogic JMS Wrappers 1
Declaring a JMSContext Object Using @Inject Annotation 1
Specifying a Lookup Name in JMSContext Injection 2
Determining the Authentication Type for JMSContext Injection 2
Declaring JMS Objects as Resources In the EJB or Servlet Deployment Descriptors 3
Declaring a Wrapped JMS Factory using Deployment Descriptors 3
Declaring JMS Destinations using Deployment Descriptors 4
Referencing a Packaged JMS Application Module In Deployment Descriptor Files 5
Referencing Application Modules in a weblogic-application.xml Descriptor 5

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

Referencing JMS Resources in a WebLogic Application 6
Referencing JMS Resources in a Jakarta EE Application 6
Declaring JMS Destinations and Connection Factories Using Annotations 6
Injecting Resource Dependency into a Class 6
Non-Injected EJB 3.0 Resource Reference Annotations 7
Avoid Transactional XA Interfaces 8
Disabling Wrapping and Pooling 8
What's Happening Under the JMS Wrapper Covers 8
Automatically Enlisting Transactions 8
Container-Managed Security 9
Connection Testing 9
Jakarta EE Compliance 9
Pooled JMS Connection Objects 10
Improving Performance Through Pooling 10
Speeding Up JNDI Lookups by Pooling Session Objects 10
Speeding Up Object Creation Through Caching 11
Enlisting the Proper Transaction Mode 11
Simplified Access to Foreign JMS Providers 11
Examples of JMS Wrapper Functions 12
Stateless Session Bean Example 12
ejb-jar.xml 12
weblogic-ejb-jar.xml 13
PoolTest.java 13
PoolTestHome.java 13
PoolTestBean.java 14
Sending a JMS Message in a Jakarta EE Container 15
Using comp/env 15
Dependency Injection 16
EJB 3.0 Wrapper Without Injection 17

4 Understanding the Simplified API Programming Model

About JMS 2.0 Simplified API 1
New Interfaces in the Simplified JMS API 2
JMSContext 2
JMSProducer 2
JMSConsumer 2
New Methods to Simplify Messaging in JMS 2.0 2
Method to Extract the Body Directly from a Message 3
Method to Receive a Message Body Directly 3
Method to Create a Session 3

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

5 Developing a Basic JMS Application

Importing Required Packages 1
Setting Up a JMS Application 1
Using a Simplified API to Set Up a JMS Application 2
Look Up a Connection Factory in JNDI 2
Look Up a Queue or Topic 3
Create a JMSContext Object 3
Create JMSProducer and JMSConsumer Objects 3
Sending and Receiving Messages using the Simplified API 4
Using the Classic API to Set Up a JMS Application 4
Step 1: Look Up a Connection Factory in JNDI 5
Step 2: Create a Connection Using the Connection Factory 6
Step 3: Create a Session Using the Connection 7
Step 4: Look Up a Destination (Queue or Topic) 8
Step 5: Create Message Producers and Message Consumers 9
Step 6a: Create the Message Object (Message Producers) 11
Step 6b: Optionally Register an Asynchronous Message Listener 12
Step 7: Start the Connection 13
Example: Setting Up a Point-to-Point JMS Application Using the Classic API 13
Example: Setting Up a Publish-Subscribe JMS Application Using the Classic API 16
Sending Messages 18
Sending Messages Using the Simplified JMS API 18
Sending Messages Using the Classic JMS API 19
Create a Message Object 19
Define a Message 19
Send the Message to a Destination Using MessageProducer 20
Sending a Message Asynchronously 21
Setting JMSProducer and MessageProducer Attributes 22
Example: Sending Messages Within a Point-to-Point Application 23
Example: Sending Messages Within a Publish/Subscribe Application 23
Receiving Messages 24
Receive Messages Asynchronously Using the Simplified API 24
Receiving Messages Asynchronously using the Classic API 24
Asynchronous Message Pipeline 25
Configuring a Message Pipeline 25
Behavior of Pipelined Messages 25
Receive Messages Synchronously Using the Simplified API 25
Receiving Messages Synchronously Using the Classic API 26
Example: Receiving Messages Synchronously Within a PTP Application 26
Example: Receiving Messages Synchronously Within a Pub/Sub Application 26
Use Prefetch Mode to Create a Synchronous Message Pipeline 27

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

Recovering Received Messages 27
Acknowledging Received Messages 28
Releasing Object Resources 28

6 Managing Your Applications

Managing Rolled Back, Recovered, Redelivered, or Expired Messages 1
Setting a Redelivery Delay for Messages 1
Setting a Redelivery Delay 1
Overriding the Redelivery Delay on a Destination 2
Setting a Redelivery Limit for Messages 2
Ordered Redelivery of Messages 2
Required Message Pipeline Setting for the Messaging Bridge and MDBs 3
Performance Limitations 3
Handling Expired Messages 3
Setting Message Delivery Times 4
Setting a Delivery Time on Producers 4
Setting a Delivery Time on Messages 4
Overriding a Delivery Time 5
Interaction with the Time-to-Live Value 5
Setting a Scheduled Time-to-Deliver Override 5
JMS Schedule Interface 7
Managing Connections 8
Defining a Connection Exception Listener 8
Accessing Connection Metadata 8
Starting, Stopping, and Closing a Connection 9
Managing Sessions 10
Defining a Session Exception Listener 10
Closing a Session 11
Managing Destinations 11
Dynamically Creating Destinations 12
Dynamically Deleting Destinations 12
Required Conditions for Deleting Destinations 12
What Happens when a Destination Is Deleted 12
Message Timestamps for Troubleshooting Deleted Destinations 13
Deleted Destination Statistics 14
Using Temporary Destinations 14
Creating a Temporary Queue 14
Creating a Temporary Topic 14
Deleting a Temporary Destination 15
Setting Up Durable Subscriptions 15
Defining the Persistent Store 15

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

Setting the Client ID Policy 15
Defining the Client ID 16
Creating a Sharable Subscription Policy 17
Creating Subscribers for a Durable Subscription 18
Using JMS 2.0 API 18
Using JMS 1.1 API 18
Best Practice: Always Close Failed JMS ClientIDs 19
Deleting Durable Subscriptions 19
Modifying Durable Subscriptions 19
Managing Durable Subscriptions 20
Setting and Browsing Message Header and Property Fields 20
Setting Message Header Fields 20
Setting Message Property Fields 22
Browsing Header and Property Fields 24
Filtering Messages 26
Defining Message Selectors Using SQL Statements 26
Defining XML Message Selectors Using XML Selector Method 27
Displaying Message Selectors 28
Indexing Topic Subscriber Message Selectors to Optimize Performance 28
Sending XML Messages 29
WebLogic XML APIs 29
Using a String Representation 29
Using a DOM Representation 30

7 Using JMS Module Helper to Manage Applications

Configuring JMS System Resources Using JMSModuleHelper 1
Configuring JMS Servers and Store-and-Forward Agents 1
JMSModuleHelper Sample Code 2
Creating a JMS System Resource 2
Deleting a JMS System Resource 3
Security Considerations for Anonymous Users 4
Best Practices When Using JMSModuleHelper 4

8 Using Multicasting with WebLogic JMS

Benefits of Using Multicasting 1
Limitations of Using Multicasting 1
Using WebLogic Server Unicast 1
Configuring Multicasting for WebLogic Server 2
Prerequisites for Multicasting 2
Step 1: Set Up the JMS Application, Multicast Session and Topic Subscriber 3

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

Step 2: Set Up the Message Listener 3
Dynamically Configuring Multicasting Configuration Attributes 4
Example: Multicast Time-to-Live 5

9 Using Distributed Destinations

What Is a Distributed Destination? 1
Why Use a Distributed Destination 1
Creating a Distributed Destination 1
Types of Distributed Destinations 1
Uniform Distributed Destinations 1
Weighted Distributed Destinations 2
Using Distributed Destinations 2
Using Distributed Queues 2
QueueSenders 3
QueueReceivers 3
QueueBrowsers 3
Using Replicated Distributed Topics 4
TopicPublishers 4
TopicSubscribers 5
Deploying Message-Driven Beans on a Distributed Topic 6
Using Partitioned Distributed Topics 6
Accessing Distributed Destination Members 7
Distributed Destination Failover 7
Using Message-Driven Beans with Distributed Destinations 7
Common Use Cases for Distributed Destinations 7
Maximizing Production 8
Maximizing Availability 8
Using Queues 8
Using Topics 9
Stuck Messages 10

10 Using the Message Unit-of-Order

What is Message Unit-Of-Order? 1
Understanding Message Processing with Unit-of-Order 1
Message Processing According to the JMS Specification 1
Message Processing with Unit-of-Order 1
Message Delivery with Unit-of-Order 2
Message Unit-of-Order Case Study 3
Joe Orders a Book 3
What Happened to Joe's Order 3

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

How Message Unit-of-Order Solves the Problem 4
How to Create a Unit-of-Order 5
Creating a Unit-of-Order Programmatically 5
Unit-of-Order Naming Rules 6
Getting the Current Unit-of-Order 6
Message Unit-of-Order Advanced Topics 7
What Happens When a Message Is Delayed During Processing? 7
What Happens When a Filter Makes a Message Undeliverable 7
What Happens When Destination Sort Keys Are Used 7
Using Unit-of-Order with Distributed Destinations 8
Using the Path Service 8
Using Hash-Based Routing 8
Using Unit-of-Order with Topics 8
Unit-of-Order and Distributed Topics 8
Unit-of-Order, Topics, and Message Driven Beans 9
Using Unit-of-Order with JMS Message Management 9
Using Unit-of-Order with WebLogic Store-and-Forward 10
Using Unit-of-Order with WebLogic Messaging Bridge 10
Limitations of Message Unit-of-Order 10

11 Using Unit-of-Work Message Groups

What Are Unit-of-Work Message Groups? 1
Understanding Message Processing with Unit-of-Work 1
Basic UOW Terminology 1
Rules For Processing UOW Messages 2
Message Unit-of-Work Case Study 3
How to Create a Unit-of-Work Message Group 4
How to Write a Producer to Set UOW Message Properties 5
Example UOW Producer Code 5
UOW Exceptions 6
How to Write a UOW Consumer/Producer For an Intermediate Destination 6
Configuring Terminal Destinations 7
UOW Message Routing for Terminal Distributed Destinations 8
How to Write a UOW Consumer for a Terminal Destination 8
Message Unit-of-Work Advanced Topics 9
Message Property Handling 9
System-Generated Properties 9
Final Component Message Properties 9
Component Message Heterogeneity 10
JMSReplyTo Header 10
UOW and Uniform Distributed Destinations 10

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

UOW and Store-and-Forward Destinations 10
Limitations of UOW Message Groups 10

12 Using Transactions with WebLogic JMS

Overview of Transactions 1
Using JMS Transacted Sessions 2
Step 1: Set Up JMS Application, Creating Transacted Session 3
Step 2: Perform Desired Operations 3
Step 3: Commit or Roll Back the JMS Transacted Session 3
Using JTA User Transactions 4
Step 1: Set Up JMS Application, Creating Non-Transacted Session 5
Step 2: Look Up the User Transaction in JNDI 6
Step 3: Start the JTA User Transaction 6
Step 4: Perform Desired Operations 6
Step 5: Commit or Roll Back the JTA User Transaction 6
JTA Transactions Using Message-Driven Beans 7
Example: JMS and EJB in a JTA User Transaction 7
Step 1 Set Up the JMS Application 8
Step 2 Look Up the User Transaction 8
Step 3 Start the JTA User Transaction 8
Step 4 Perform the Desired Operations 8
Step 5 Commit the JTA User Transaction 8
Using Cross-Domain Security 8

13 Developing Advanced Pub/Sub Applications

Overview of Advanced High Availability Concepts 1
WebLogic Messaging High Availability Features 1
Application Design Limitations When Using Replicated Distributed Topics 2
Advanced Topic Features 2
Advanced Topic Messaging Features for High Availability 3
Shared Subscriptions and Client ID Policy 3
What Is the Subscription Key 3
Configuring a Shared Subscription 3
How Sharing a Non-Durable Subscription Works 4
How a Shared Subscription Policy for a Non-Durable Subscription Is Determined 4
How a Non-Durable Subscription Is Closed 4
How Sharing a Durable Subscription Works 5
How a Shared Subscription Policy for a Durable Subscription Is Determined 5
How to Unsubscribe a Durable Subscription 5
Considerations When Unsubscribing a Durable Subscriber 6

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

Managing Durable Subscriptions 6
Design Strategies When Using Topics 7
One-Copy-Per-Instance Design Strategy 7
One-Copy-Per-Application Design Strategy 8
Considerations When Using Standard Shared Subscriptions 8
Replacing a Replicated Distributed Topic 8
Reasons for Replacing a Replicated Distributed Topic 8
Important Prerequisites Before Replacing an RDT 9
Replacing an RDT with a Standalone Topic 9
Replacing an RDT with a PDT 9
Best Practices for Distributed Topics 10

14 Recovering from a Server Failure

Automatic JMS Client Failover 1
Automatic Reconnect Limitations 1
Automatic Failover for JMS Producers 2
Sample Producer Code 2
Reusable ConnectionFactory Objects 2
Reusable Destination Objects 3
Reconnected Connection Objects 3
Reconnected Session Objects 4
Reconnected MessageProducer Objects 4
Configuring Automatic Failover for JMS Consumers 5
Sample Consumer Client Code 5
Configuring Automatic Client Refresh Options 6
Common Cases for Reconnected Consumers 6
Special Cases for Reconnected Consumers 7
Explicitly Disabling Automatic Failover on JMS Clients 8
Programmatically 9
Administratively 9
Best Practices for JMS Clients Using Automatic Failover 9
Always Catch Exceptions 9
Use Transactions to Group Message Work 9
JMS Clients Should Always Call the close() Method 10
Manually Migrating JMS Data to a New Server 10

15 Understanding WebLogic JMS Security

Securing WebLogic JMS Resources 1
Understanding Thread-Based Security on Clients and Servers 1
Thread-Based Security for Server Applications 2

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

Thread-Based Security for Client Applications 2
Understanding Object-Based Security 3
Enabling Object-Based Security on Clients 3
Object-Based Security Limitations on Clients 4
Enabling Object-Based Security on Server Applications 4
Object-Based Security for Inbound JMS Applications 4
Object-Based Security for Outbound JMS Applications 5
Understanding Cross-Domain Security 6
Cross-Domain Security Guidelines 6
Programming Pattern for a Single JMS Client Communicating With Two WebLogic Domains 7
Programming Patterns for Using a Foreign JMS Server Between Two WebLogic Domains 9

16 WebLogic JMS C API

What Is the WebLogic JMS C API? 1
System Requirements 1
Design Principles 2
Java Objects Map to Handles 2
Thread Utilization 2
Exception Handling 2
Type Conversions 2
Integer (int) 3
Long (long) 3
Character (char) 3
String 3
Memory Allocation and Garbage Collection 3
Closing Connections 4
Helper Functions 4
Security Considerations 4
Implementation Guidelines 4
Client Packaging Requirements 5
Workarounds for Client Failure Thread Detach Issue 5

A FAQs: Integrating Remote JMS Providers

Understanding JMS and JNDI Terminology A-1
Understanding Transactions A-2
How to Integrate with a Remote Provider A-3
Best Practices When Integrating with Remote Providers A-5
Using Foreign JMS Server Definitions A-6
Using EJB/Servlet JMS Resource References A-6

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

Using WebLogic Store-and-Forward A-7
Using WebLogic JMS SAF Client A-8
Using a Messaging Bridge A-8
Using Message-Driven Beans A-9
Using AQ JMS A-10

B How to Look Up a Destination

Use a JNDI Name B-1
Use a Create Destination Identifier B-1
- Default WebLogic CDI Syntax B-2
- Custom WebLogic CDI Syntax B-2
- Server Affinity When Looking Up Destinations B-2

Examples of Syntax Used to Look Up Destinations B-2
- Non-Distributed Destinations B-3
  - JNDI Syntax for Non-Distributed Destinations B-3
  - CDI Syntax for Non-Distributed Destinations B-3
- Uniform Distributed Destinations B-3
  - JNDI Syntax for UDDs B-3
  - CDI Syntax for UDDs B-4
- Weighted Distributed Destinations B-5
  - JNDI Syntax for WDDs B-5
  - CDI Syntax for WDDs B-5

C Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API

Introduction C-1
Controlling DD Producer Load Balancing C-1
- Basic JMS C-2
- Senders to Distributed Queues (DQs) and Partitioned Distributed Topics (PDTs) C-2
- Senders to Replicated Distributed Topics (RDTs) C-2

Using the JMS Destination Availability Helper API C-2
- Overview C-2
- General Flow C-3
- Handling the weblogic.jms.extensions.DestinationDetail C-4
- Best Practices for Consumer Containers C-4
  - When to Register and Unregister C-4
  - URL Handling C-4
  - Failure Handling C-5
  - JNDI Context Handling C-5
  - JMS Connection Handling C-5

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

Interoperability Guidelines C-6
API Availability C-6
Foreign Contexts C-6
Destination Type Support C-6
Unavailable Notifications C-6
Historical Behavior for Distributed Destinations Before WebLogic Server 9.0 C-6
Historical Behavior for Distributed Topics Before WebLogic Server 10.3.4 C-7
DestinationDetail Fields C-7
Security Considerations C-7
WebLogic Server Security Model C-7
Passing Credentials Between Threads C-7
Managing Cross-Domain Security C-8
Authentication of Users C-8
Securing Destinations C-9
Securing Wire Data C-9
Transaction Considerations C-9
Strategies for Uniform Distributed Queue Consumers C-10
General Strategies C-10
Best Practice for Local Server Consumers C-10
Strategies for Subscribers on Uniform Distributed Topics C-11
One Copy Per Instance C-11
General Pattern Design Strategy for One Copy Per Instance C-11
Best Practice for Local Server Consumers using One Copy Per Instance C-12
One Copy Per Application C-12
General Pattern Design Strategy for One Copy Per Application C-12
Best Practice for Local Server Consumers Using One Copy Per Application C-13

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

Preface

This document is a resource for software developers who want to develop and configure applications that include WebLogic Server Java Message Service (JMS).

Audience

This document contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Server JMS for a particular application.

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

This document does not address production phase administration, monitoring, or performance tuning JMS topics.

You should be familiar with Jakarta EE and JMS concepts. This document emphasizes the value-added features provided by WebLogic Server JMS and key information about how to use WebLogic Server features and facilities to get a JMS application up and running.

Documentation Accessibility

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

Access to Oracle Support

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

Diversity and Inclusion

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

Related Documentation

This document contains JMS-specific design and development information.

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

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

- *Administering JMS Resources for Oracle WebLogic Server* for information about configuring and managing JMS resources.
- *Administering the Store-and-Forward Service for Oracle WebLogic Server* for information about the benefits and usage of the Store-and-Forward service with WebLogic JMS.
- *Administering the WebLogic Persistent Store* for information about the benefits and usage of the system-wide WebLogic Persistent Store.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.

Samples and Tutorials for the JMS Developer

In addition to this document, Oracle provides a variety of code samples and tutorials for JMS developers. The samples and tutorials illustrate WebLogic Server JMS in action, and provide practical instructions about how to perform key JMS development tasks.

Oracle recommends that you run some or all of the JMS examples before developing your own JMS applications.

For more information, see Sample Applications and Code Examples

New and Changed WebLogic Server Features

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

Conventions

The following text conventions are used in this document:

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

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

1

Understanding WebLogic JMS

Learn about the different Java Message Service (JMS) concepts and features, and understand how they work with other application objects and WebLogic Server.
This chapter assumes familiarity with Java programming and the basic concepts of Jakarta Messaging. WebLogic JMS supports the Jakarta Messaging 3.1 API in the jakarta.jms package. This guide uses JMS as shorthand for the messaging API and for WebLogic JMS.

Overview of Jakarta Messaging and WebLogic JMS

WebLogic JMS is an enterprise-class messaging system that is tightly integrated into the WebLogic Server platform.

WebLogic JMS implements Jakarta Messaging 3.1 and provides WebLogic JMS extensions in addition to the standard API. See the Jakarta Messaging 3.1 specification and the WebLogic JMS Extensions API reference.

What Is Jakarta Messaging?

An enterprise messaging system enables applications to communicate with one another through the exchange of messages. A message is a request, report, and/or event that contains information needed to coordinate communication between different applications. A message provides a level of abstraction between application code and the underlying messaging system.

Jakarta Messaging is the standard Java API for enterprise messaging. It continues the Java Message Service (JMS) API and provides interfaces for creating, sending, and receiving messages. Specifically, Jakarta Messaging:

- Enables Java applications sharing a messaging system to exchange messages
- Simplifies application development by providing a standard interface for creating, sending, and receiving messages

The following figure illustrates WebLogic JMS messaging.

Figure 1-1 WebLogic JMS Messaging



As shown in the figure, WebLogic JMS accepts messages from *producer* applications and delivers them to *consumer* applications.

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

Messaging Specification and API Version

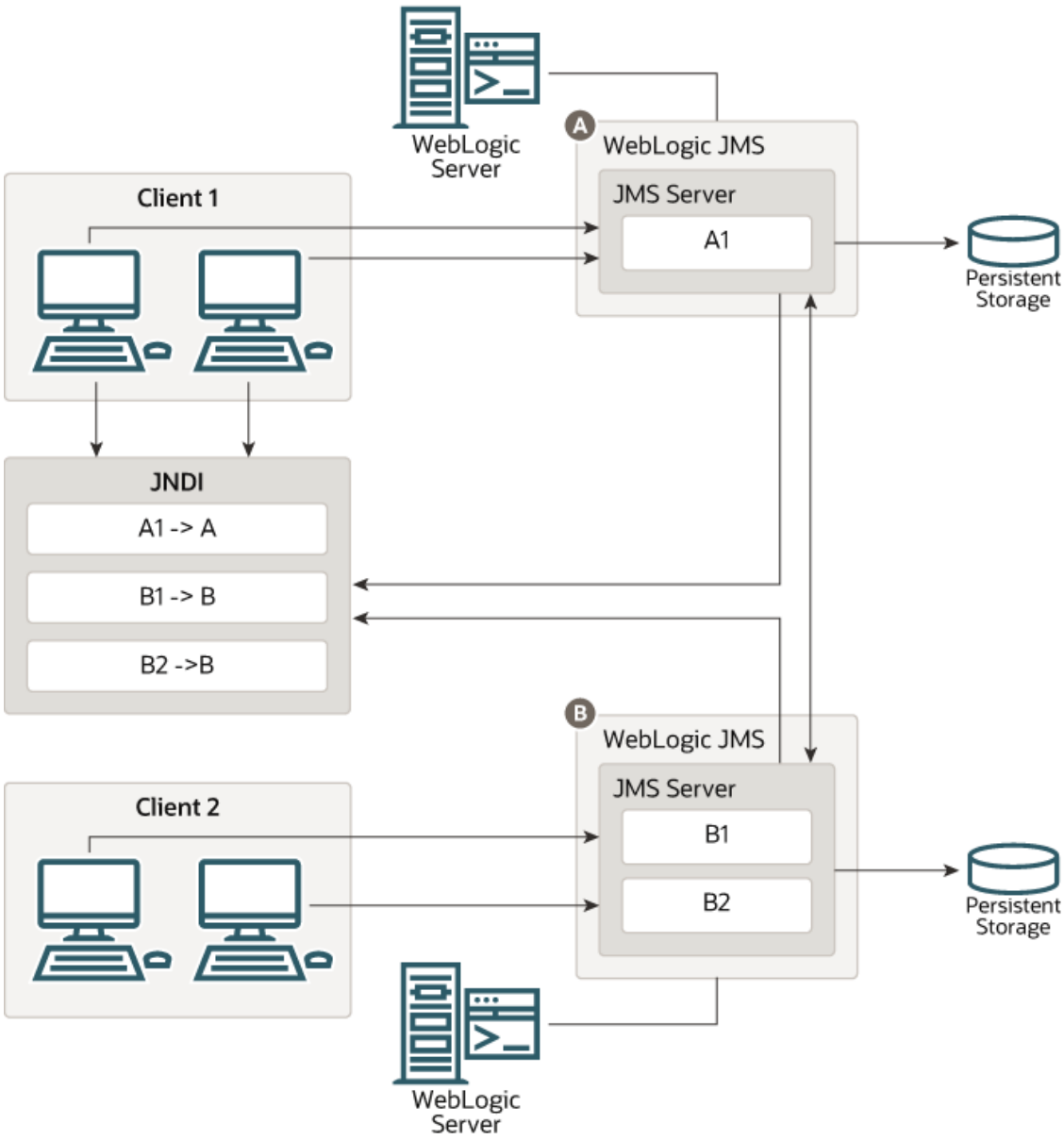
This release uses Jakarta Messaging 3.1. Its jakarta.jms package includes the classic API, the simplified API, and the legacy queue-specific and topic-specific interfaces. Historical references to JMS 1.1 and JMS 2.0 explain the origins of these programming models; they do not change the API namespace used by this release.

See the Jakarta Messaging 3.1 specification.

WebLogic JMS Architecture

The following figure illustrates the WebLogic JMS architecture.

Figure 1-2 WebLogic JMS Architecture



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

The major components of the WebLogic JMS Server architecture include:

- JMS server: a service that hosts queue and topic destinations and manages their messages and durable-subscription state. Destination configuration is defined in JMS modules. A module can contain resources with different deployment targets, subject to the targeting rules for each resource type. A WebLogic Server instance can host multiple JMS server instances. When persistence is enabled, a JMS server uses its configured persistent store or, where applicable, the hosting server instance's default store.
- JMS connection hosts and connection factories: any WebLogic server in a cluster can act as a JMS connection host for JMS applications. A JMS application gains access to WebLogic JMS by (a) obtaining a connection factory reference from JNDI, (b) obtaining a connection from this factory, and finally (c) using the connection to send or receive messages. JMS messages flow from an application, through its connection host, and then to any destination on a JMS server that is in the same cluster as the connection host. An application can use either default connection factories or custom connection factories that are configured using a JMS module.
- JMS destinations: hold JMS messages and are hosted on JMS servers. WebLogic JMS applications typically obtain JMS destination references via JNDI and then send and receive messages to these destinations using their respective JMS connections. A single logical WebLogic destination can be configured to be distributed across multiple JMS servers within the same cluster. A WebLogic JMS client can transparently communicate with any WebLogic JMS destination that is hosted in the same cluster as the client's connection host.
- JMS modules: contain configuration resources, such as standalone queue and topic destinations, distributed destinations, and connection factories, and are defined by XML documents that conform to the `http://xmlns.oracle.com/weblogic/weblogic-jms/1.8/weblogic-jms.xsd` schema.
- Client JMS applications: either produce messages to destinations or consume messages from destinations.
- JNDI (Java Naming and Directory Interface): provides a lookup facility for JMS connection factories and destinations.
- WebLogic persistent storage: a server instance's default store, a user-defined file store, or a user-defined JDBC-accessible store for storing persistent message data.

Understanding the Messaging Models

JMS supports two messaging models: point-to-point (PTP) and publish/subscribe.

The messaging models are similar, except for the following differences:

- Point-to-point messaging sends messages to a queue. Each delivery goes to one eligible consumer of that queue.
- Publish/subscribe messaging sends messages to a topic. Each eligible subscription can receive its own copy of a message.

The `jakarta.jms.Queue` and `jakarta.jms.Topic` interfaces both extend `jakarta.jms.Destination`. They identify the queue or topic used by the messaging API.

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

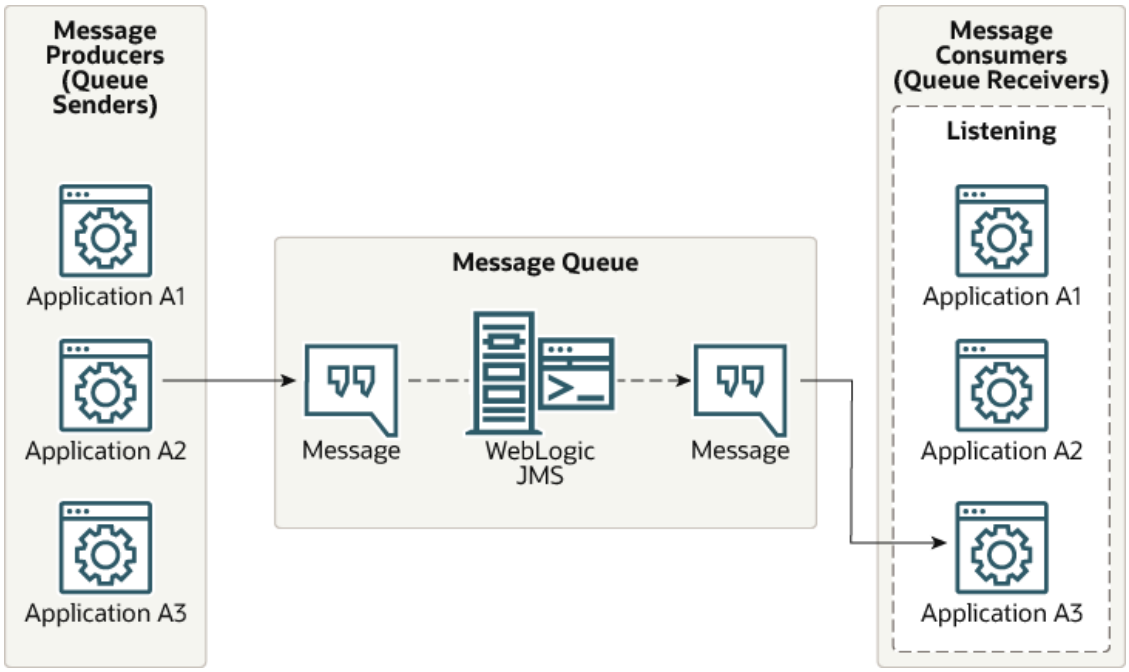
> **Note**
>
> The terms *producer* and *consumer* are used as generic descriptions of applications that send and receive messages, respectively, in either messaging model. For each specific messaging model, however, unique terms specific to that model are used when referring to producers and consumers.

Point-to-Point Messaging

The point-to-point (PTP) messaging model enables one application to send a message to another. PTP messaging applications send and receive messages using named queues. A *queue sender* (producer) sends a message to a specific queue. A *queue receiver* (consumer) receives messages from a specific queue.

The following figure illustrates PTP messaging.

Figure 1-3 Point-to-Point (PTP) Messaging



A queue can have multiple producers and consumers. WebLogic JMS selects an eligible consumer for each delivery, taking account of message selectors, consumer capacity, and applicable message-group rules. Applications must not rely on a fixed first-come-first-served order among consumers.

If no eligible consumer is available, a message can wait on the queue, subject to expiration, administrative removal, and the failure-recovery guarantees of its delivery mode. An unacknowledged message can be redelivered, including to a different eligible consumer.

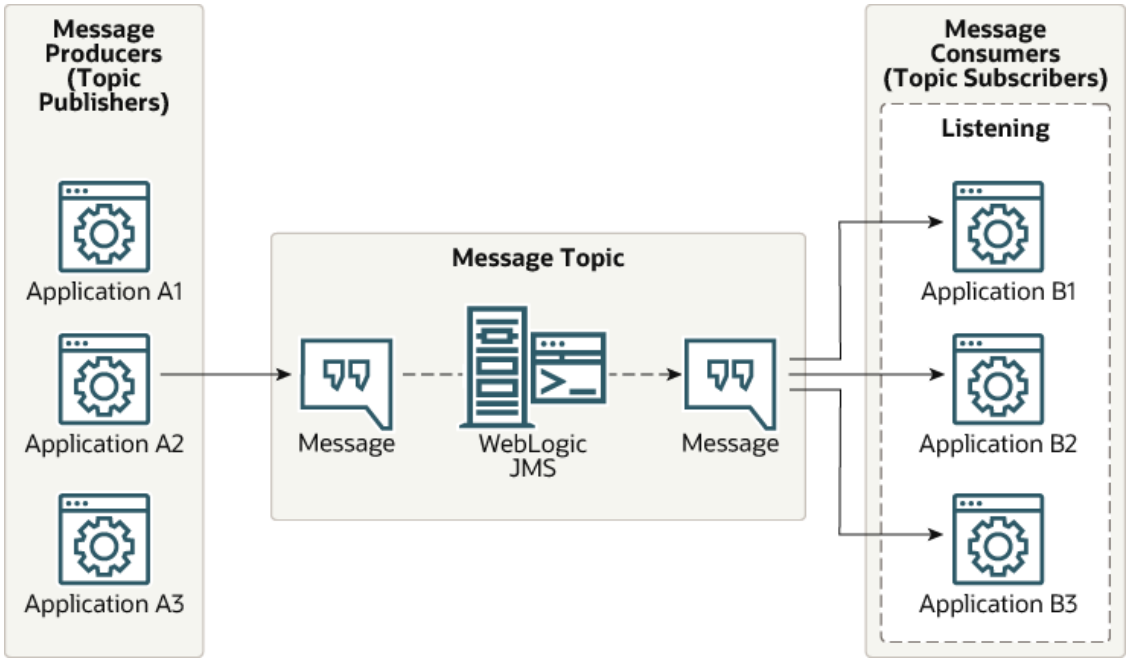
Copyright © 2007, 2026, Oracle and/or its affiliates.

Publish/Subscribe Messaging

In publish/subscribe messaging, producers publish messages to a topic. Consumers receive messages through subscriptions to that topic. A producer does not need to create a subscription to publish messages.

The following figure illustrates publish/subscribe messaging.

Figure 1-4 Publish/Subscribe Messaging



Each eligible subscription receives its own copy of a topic message, subject to its message selector and applicable options. A shared subscription can have multiple consumers, but each message in that subscription is delivered to one consumer at a time.

A durable subscription can retain eligible messages while its consumers are inactive. A nondurable subscription does not provide that offline-retention guarantee. Message retention also depends on acknowledgement, transaction completion, expiration, and delivery-mode guarantees. See Using Durable Subscriptions.

Message Persistence

A message's delivery mode determines the provider-failure protection requested for that message. It does not guarantee that application processing occurs only once. See Message Delivery Mode in the Jakarta Messaging 3.1 specification.

- **PERSISTENT:** With persistence enabled, WebLogic JMS uses persistent storage to protect pending messages against provider failure, subject to the messaging contract. Configure the JMS server's persistent store and applicable targeting requirements. If no custom store is configured, the hosting WebLogic Server instance's default store is used where that configuration is supported.

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

- **NON_PERSISTENT:** Messages do not have the same provider-failure recovery guarantee and can be lost on failure. WebLogic JMS can page message data to disk to manage memory; paging does not make a nonpersistent message durable.

Acknowledgement and transaction completion govern removal and redelivery. Recovery or rollback can cause an unacknowledged message to be delivered again. Closing a connection does not guarantee retention for a nondurable topic subscription. Expiration and administrative removal also apply.

For information about configuring persistent stores, see *Administering the WebLogic Persistent Store*.

Understanding the JMS API

The jakarta.jms API provides interfaces for connecting to a messaging provider and sending and receiving messages.

Jakarta Messaging provides a classic API based on `Connection` and `Session` and a simplified API based on `JMSContext`, `JMSProducer`, and JMSConsumer. It also retains the legacy queue-specific and topic-specific interfaces. See the Jakarta Messaging 3.1 API reference for the standard interfaces and WebLogic JMS Extensions for the WebLogic-specific APIs.

The following table introduces the APIs discussed in this chapter.

Table 1-1 JMS APIs

API	Description
ConnectionFactory	Creates Connection objects for the classic API and JMSContext objects for the simplified API. Applications commonly obtain a connection factory through JNDI.
JMSContext	Encapsulates the functionality of two objects, Connection and Session, in a single object.
Connection	Represents an open communication channel to the messaging system. A connection is used to create sessions.
Session	Provides a single-threaded context for creating, sending, and receiving messages, with acknowledgement and transaction semantics.
Destination	Identifies the queue or topic used by messaging operations.
MessageProducer and MessageConsumer	Provides the interface for sending and receiving messages. Message producers send messages to a queue or topic. Message consumers receive messages from a queue or topic.
Messages	Encapsulates information to be sent or received.
ServerSessionPoolFactory[1]	Encapsulates configuration information for a server-managed pool of message consumers. The server session pool factory is used to create server session pools.
ServerSessionPool[2]	Provides a pool of server sessions that can be used to process messages concurrently for connection consumers.
ServerSession[3]	Associates a thread with a JMS session.

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

Table 1-1 (Cont.) JMS APIs

API	Description
ConnectionConsumer[4]	Specifies a consumer that retrieves server sessions to process messages concurrently.

1 WebLogic-specific factory interface for the legacy server-session-pool facility.
2 Supports an optional JMS interface for processing multiple messages concurrently.
3 Supports an optional JMS interface for processing multiple messages concurrently.
4 Supports an optional JMS interface for processing multiple messages concurrently.

For information about configuring JMS resources, see Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*. The procedure for setting up a JMS application is presented in Setting Up a JMS Application.

ConnectionFactory

`ConnectionFactory` encapsulates connection configuration information and enables applications to create a `Connection` or `JMSContext`. A connection factory supports concurrent use, enabling multiple threads to access the object simultaneously. You can use the pre configured default connection factories provided by WebLogic JMS, or you can configure one or more connection factories to create connections with predefined attributes that suit your application.

Using the Default Connection Factories

WebLogic Server provides the Jakarta EE default JMS connection factory at `java:comp/DefaultJMSConnectionFactory`. See Using the Default JMS Connection Factory in Administering JMS Resources for Oracle WebLogic Server.

WebLogic JMS defines two default connection factories, which you can look up using the following JNDI names:

- `weblogic.jms.ConnectionFactory`
- `weblogic.jms.XAConnectionFactory`

You only need to create a user-defined connection factory if the settings of the default factories are not suitable for your application. The main difference between the preconfigured settings for the default connection factories is the default value for the "XA Connection Factory Enabled" attribute which is used to enable JTA transactions, as shown in the following table.

Table 1-2 XA Transaction Settings for Default Connection Factories

Default Connection Factory	XA Connection Factory Enabled setting is
`weblogic.jms.ConnectionFactory`	False
`weblogic.jms.XAConnectionFactory`	True

An XA factory is required for JMS applications to use JTA user transactions, but is not required for transacted sessions. For more information about using transactions with WebLogic JMS, see Using Transactions with WebLogic JMS.

All other default factory configuration attributes are set to the same default values as a user-defined connection factory.

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

The WebLogic default connection factories are deployed by the server rather than targeted individually. You can disable these WebLogic defaults on a per-server basis with the JMS Default Connection Factories Enabled setting. The Jakarta EE default connection factory at `java:comp/DefaultJMSConnectionFactory` remains available when the WebLogic defaults are disabled.

To deploy a connection factory on specific independent servers, on specific servers within a cluster, or on an entire cluster, you must configure a new connection factory and specify the appropriate target, as explained in Connection Factory Configuration in *Administering JMS Resources for Oracle WebLogic Server*.

> **Note**
>
> For backward compatibility, WebLogic JMS still supports two deprecated default connection factories. The JNDI names for these factories are `jakarta.jms.QueueConnectionFactory` and `jakarta.jms.TopicConnectionFactory`.

Configuring and Deploying Connection Factories

A system administrator can define and configure one or more connection factories to create connections with predefined attributes and WebLogic Server will add them to the JNDI space during startup. The application then retrieves a connection factory using WebLogic JNDI. Any user-defined connection factories must be uniquely named.

For information about configuring connection factories, see Configure Resources for JMS System Modules in the Oracle WebLogic Remote Console Online Help.

A system administrator establishes cluster-wide, transparent access to JMS destinations from any server in the cluster by targeting to the cluster or by targeting to one or more server instances in the cluster. This way, each connection factory can be deployed on multiple WebLogic Server instances. For more information about JMS clustering, refer to Configuring Advanced WebLogic JMS Resources in *Administering JMS Resources for Oracle WebLogic Server*.

The ConnectionFactory Interface

The `ConnectionFactory` interface defines methods that create `Connection` objects for the classic API and `JMSContext` objects for the simplified API. Both APIs support queue and topic messaging. A connection factory supports concurrent use by multiple threads. The domain-specific `QueueConnectionFactory` and `TopicConnectionFactory` interfaces remain available for compatibility.

The following table describes the domain-specific connection factory interfaces.

Table 1-3 Domain-Specific ConnectionFactory Interfaces

Interface	In Messaging Model	Is Used to Create
`QueueConnectionFactory`	PTP	`QueueConnection` to a JMS PTP provider.
`TopicConnectionFactory`	Publish/Subscribe	`TopicConnection` to a JMS Publish/Subscribe provider.

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

To learn how to use the `ConnectionFactory` interface within an application, see Developing a Basic JMS Application, or the `jakarta.jms.ConnectionFactory` Javadoc at `jakarta.jms.ConnectionFactory`.

JMSContext

`JMSContext` is the main interface introduced in the simplified API for JMS 2.0. For more information about this interface, see New Interfaces in the Simplified JMS API.

Connection

A `Connection` represents an open communication channel between an application and the messaging system, and is used to create a `Session` (see Session) for producing and consuming messages. A connection creates server-side and client-side objects that manage the messaging activity between an application and JMS. A connection may also provide user authentication.

A `Connection` is created by `ConnectionFactory` (see ConnectionFactory), obtained through a JNDI lookup.

Due to the resource overhead associated with authenticating users and setting up communications, most applications establish a single connection for all messaging. WebLogic can multiplex JMS traffic over shared T3 or T3S transport connections, so each JMS `Connection` does not require a separate TCP/IP connection. Additional transport connections can be created as needed when contacting other servers, using different network channels or protocols, or reestablishing a failed transport connection. Servlets and other server-side objects can also obtain JMS Connections.

By default, a connection is created in stopped mode. For information about how and when to start a stopped connection, see Starting, Stopping, and Closing a Connection.

Connections support concurrent use, enabling multiple threads to access the object simultaneously.

> **Note**
>
> Use `Connection` for both queue and topic messaging. The domain-specific `QueueConnection` and `TopicConnection` interfaces remain available for compatibility.

The following table describes the domain-specific connection interfaces.

Table 1-4 Domain-Specific Connection Interfaces

Interface	In Messaging Model	Is Used to Create
`QueueConnection`	PTP	`QueueSessions`, and consists of a connection to a JMS PTP provider created by `QueueConnectionFactory`.
`TopicConnection`	Pub/sub	`TopicSessions`, and consists of a connection to a JMS publish/subscribe provider created by `TopicConnectionFactory`.

To learn how to use the `Connection` interface within an application, see Developing a Basic JMS Application, or the `jakarta.jms.Connection` Javadoc at `jakarta.jms.Connection`.

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

Session

A `Session` object defines a serial order for the messages produced and consumed, and can create multiple message producers and message consumers. The same thread can be used for producing and consuming messages. If you want an application to have a separate thread for producing and consuming messages, then the application should create a separate session for each function.

A Session is created by `Connection` (see Connection).

WebLogic JMS Session Guidelines

A generic `Session` can create producers and consumers for both queues and topics, subject to the restrictions of the destination and acknowledgement mode. The domain-specific `QueueSession` and `TopicSession` interfaces retain their queue-only and topic-only restrictions.

Use a separate session for each thread that performs messaging concurrently. Once the connection is started, a session with registered message listeners and its producers and consumers must not be used by other application threads; this restriction also applies between callbacks. The permitted concurrent `Session.close()` and `MessageConsumer.close()` operations are exceptions. See the `jakarta.jms.Session` API documentation for the complete threading rules.

Domain-Specific Session Interfaces

The following table describes the domain-specific session interfaces.

Table 1-5 Domain-Specific Session Interfaces

Interface	In Messaging Model	Provides a Context for
`QueueSession`	PTP	Producing and consuming messages for a JMS PTP provider. Created by `QueueConnection`.
`TopicSession`	Pub/sub	Producing and consuming messages for a JMS publish/subscribe provider. Created by `TopicConnection`.

To learn how to use the Session interface within an application, see Developing a Basic JMS Application, or the `jakarta.jms.Session` at `jakarta.jms.Session`, and the weblogic.jms.extensions.WLSession Javadoc.

Non-Transacted Sessions

For a non-transacted session, Jakarta Messaging defines `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`. WebLogic JMS additionally provides `WLSession.NO_ACKNOWLEDGE` and `WLSession.MULTICAST_NO_ACKNOWLEDGE`. The following table describes these modes.

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

Table 1-6 Acknowledge Modes Used for Non-Transacted Sessions

Acknowledge Mode	Description
`AUTO_ACKNOWLEDGE`	The session acknowledges a message when a synchronous receive call successfully returns, or when the message listener's `onMessage` method successfully returns. For synchronous reception, acknowledgement does not wait for application processing performed after the receive call.
`CLIENT_ACKNOWLEDGE`	The `Session` object relies on the application to call an acknowledge method on a received message. After the method is called, the session acknowledges all messages received since the last acknowledge. This mode allows an application to receive, process, and acknowledge a batch of messages with one call.
`DUPS_OK_ACKNOWLEDGE`	The session acknowledges message delivery lazily, which can reduce acknowledgement overhead. A provider failure can cause messages to be delivered again, so applications using this mode must tolerate duplicate messages.
`NO_ACKNOWLEDGE`	No acknowledgement is required. Messages sent to a `NO_ACKNOWLEDGE` session are immediately deleted from the server. Messages received in this mode are not recovered, and as a result messages may be lost and/or duplicate message may be delivered if an initial attempt to deliver a message fails. This mode is supported for applications that do not require the quality of service provided by session acknowledge, and that do not want to incur the associated overhead. **Note:** You should avoid using this mode if your application cannot handle lost or duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.
`MULTICAST_NO_ACKNOWLEDGE`	Multicast mode with no acknowledge required. Messages sent to a `MULTICAST_NO_ACKNOWLEDGE` session share the same characteristics as `NO_ACKNOWLEDGE` mode, described previously. This mode is supported for applications that want to support multicasting, and that do not require the quality of service provided by session acknowledge. For more information on multicasting, see Using Multicasting with WebLogic JMS. **Note:** Use only with topics. You should avoid using this mode if your application cannot handle lost or duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.

Transacted Sessions

In a locally transacted session, only one local transaction is active at a time. Any number of messages sent or received during a transaction are treated as an atomic unit.

When you create a transacted session, the acknowledge mode is ignored. When an application commits a transaction, all the messages that the application received during the transaction are acknowledged by the messaging system and messages it sent are accepted for delivery. If an application rolls back a transaction, then the messages that the application received during the transaction are not acknowledged and messages it sent are discarded.

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

A local session transaction groups only that session's messaging work. To coordinate JMS operations with other transactional resources, use WebLogic Server's JTA integration. In a Jakarta EE web or EJB container, session transaction behavior follows the container's transaction rules; the application does not complete a JTA transaction by calling `Session.commit()` or `Session.rollback()`. See Using JTA User Transactions.

Destination

A `Destination` object can be either a queue or topic, encapsulating the address syntax for a specific provider. The JMS specification does not define a standard address syntax due to the variations in syntax between providers.

Similar to a connection factory, an administrator defines and configures the destination, and WebLogic Server adds it to the JNDI space during startup. Applications can also create temporary destinations that exist only for the duration of the JMS connection in which they are created.

> **Note**
>
> Administrators can also configure a distributed destination, which is a single set of destinations (queues or topics) that are accessible as a single, logical destination to a client. For more information, see Distributed Destinations.

On the client side, `Queue` and `Topic` objects are handles to the object on the server. Their methods only return their names. To access them for messaging, you create message producers and consumers that attach to them.

`Destination` objects support concurrent use. The `Queue` and `Topic` interfaces extend `jakarta.jms.Destination`; `TemporaryQueue` extends `Queue`, and `TemporaryTopic` extends Topic.

The following table describes the destination interfaces.

Table 1-7 Destination Interfaces

Interface	Messaging Model	Manages Messages for
`Queue`	PTP	JMS point-to-point provider.
`TemporaryQueue`	PTP	A temporary queue owned by the connection that created it. Only consumers created by that connection can consume from the queue. The queue is deleted when the connection closes, or earlier when the application successfully calls `delete()`.
`Topic`	Pub/sub	JMS publish/subscribe provider
`TemporaryTopic`	Pub/sub	A temporary topic owned by the connection that created it. Only consumers created by that connection can consume from the topic. The topic is deleted when the connection closes, or earlier when the application successfully calls `delete()`.

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

> **Note**
>
> An application can inspect queue messages without consuming them by creating a `QueueBrowser` from a `Session` or `QueueSession`. In WebLogic JMS, each call to `getEnumeration()` captures a snapshot of the currently visible messages that match the browser's selector. Creating the browser alone does not capture that snapshot. Browsing does not acknowledge or remove the messages. See Setting and Browsing Message Header and Property Fields.

To learn how to use the `Destination` interface within an application, see Developing a Basic JMS Application, or the `jakarta.jms.Destination` Javadoc at `jakarta.jms.Destination`.

Distributed Destinations

A distributed destination resource is a single set of destinations (queues or topics) that are accessible as a single, logical destination to a client (for example, a distributed topic has its own JNDI name). The members of the set are typically distributed across multiple servers within a cluster, with each member belonging to a separate JMS server. Distributed destinations can improve scalability and availability by distributing work across eligible members. Availability also depends on the configured JMS services, persistent stores, migration policies, and application recovery behavior. Routing to another member does not by itself make an unavailable member's retained messages accessible.

- For more information about using a distributed destination with your applications, see Using Distributed Destinations.

MessageProducer and MessageConsumer

A `MessageProducer` sends messages to a queue or topic. A `MessageConsumer` receives messages from a queue or topic. Message producers and consumers operate independently of one another. Message producers generate and send messages regardless of whether a message consumer has been created and is waiting for a message, and vice versa.

A `Session` (see Session) creates the `MessageProducers` and `MessageConsumers` that are attached to queues and topics.

The `QueueSender` and `TopicPublisher` interfaces extend `MessageProducer`, and `QueueReceiver` and `TopicSubscriber` extend MessageConsumer. These domain-specific interfaces remain available for compatibility; `MessageProducer` and `MessageConsumer` support both queues and topics.

The following table describes the domain-specific producer and consumer interfaces.

Table 1-8 Domain-Specific Producer and Consumer Interfaces

Interface	In Messaging Model	Performs this Function
`QueueSender`	PTP	Sends messages for a JMS point-to-point provider.
`QueueReceiver`	PTP	Receives messages for a JMS point-to-point provider
`TopicPublisher`	Publish/subscribe	Sends messages for a JMS Publish/subscribe provider
`TopicSubscriber`	Publish/subscribe	Receives messages for a JMS Publish/subscribe provider

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

The point-to-point model allows multiple sessions to consume from the same queue. Each message is delivered to one eligible consumer at a time. WebLogic JMS selects among consumers according to message selectors, available consumer capacity, and ordering constraints; applications must not depend on a fixed consumer-selection order. See the point-to-point messaging figure.

In the publish/subscribe model, each eligible subscription receives a copy of a message. An unshared subscription can have at most one active consumer; a shared subscription can have multiple consumers that share its messages. Subscriptions can be durable or nondurable. See the publish/subscribe messaging figure and the subscription discussion.

An application can publish and subscribe using the same connection and can receive its own publications. For subscription-creation methods that support `noLocal`, set it to `true` to exclude messages published by that connection. Shared-subscription creation methods do not provide a `noLocal` argument. See the discussion of creating topic consumers.

To learn how to use the `MessageProducer` and `MessageConsumer` interfacees within an application, see Setting Up a JMS Application, or the `jakarta.jms.MessageProducer` (at `jakarta.jms.MessageProducer`), and `jakarta.jms.MessageConsumer` (at `jakarta.jms.MessageConsumer`) Javadoc.

Messages

A `Message` encapsulates the information exchanged by applications. This information includes three components:

- Message Header Fields
- Message Property Fields
- Message Body

Message Header Fields

Every JMS message contains a standard set of header fields that is included by default and available to message consumers. Some fields can be set by the message producers.

For information about setting message header fields, see Setting and Browsing Message Header and Property Fields , or to the `jakarta.jms.Message` Javadoc at `jakarta.jms.Message`.

The following table describes the fields in the message headers and shows how values are defined for each field.

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

Table 1-9 Message Header Fields

Field	Description	Defined by
`JMSCorrelationID`	Specifies one of the following: a WebLogic `JMSMessageID` (field described later in this table), an application-specific string, or a `byte[]` array. The `JMSCorrelationID` field is used to correlate messages and is set directly on the message by the application before `send()`. There are two common applications for this field. The first application is to link messages by setting up a request/response scheme, as follows: 1. When an application sends a message, it stores the `JMSMessageID` value assigned to it. 2. When an application receives the message, it copies the `JMSMessageID` into the `JMSCorrelationID` field of a response message that it sends back to the sending application. The second application is to use the `JMSCorrelationID` field to carry any String you choose, enabling a series of messages to be linked with some application-determined value.	Application
`JMSDeliveryMode`	Specifies `PERSISTENT` or `NON_PERSISTENT`. Select the delivery mode on the producer or in the applicable send method. WebLogic JMS sets this header during the send operation and applies any configured destination override; setting the header directly on the message does not control delivery mode. With persistence enabled, persistent delivery uses persistent storage to support recovery after a server failure. Nonpersistent delivery does not provide that recovery guarantee, and messages may be lost after a failure. Message retention, acknowledgement, transactions, expiration and configured delivery policies also affect delivery and redelivery. Delivery mode alone does not guarantee that an application processes a message exactly once. See Message Persistence.	`send()` method
`JMSDeliveryTime`	Specifies the earliest time at which the message is eligible for delivery. WebLogic JMS sets this header during send processing according to the effective delivery delay, including any configured destination override. An effective delivery delay of zero makes the message eligible for immediate delivery. Configure the delay on the producer with `setDeliveryDelay()` or the WebLogic time-to-deliver extension. Setting `JMSDeliveryTime` directly on the message does not control the send. This field can be used to sort messages in a destination and to select messages. For purposes of data type conversion, the `JMSDeliveryTime` is a long integer.	`send()` method
`JMSDestination`	Specifies the destination (queue or topic) to which the message is to be delivered. This field is set when creating producer or as parameter sent by the application before `send()`. This value is overwritten by a call to `producer.send()`, setting this value directly on the message has no effect. The values set by the producer can be queried using the message supplied to `producer.send()` or when the message is received by a consumer. When a message is received, its destination value must be equivalent to the value assigned when it was sent.	`send()` method

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

Table 1-9 (Cont.) Message Header Fields

Field	Description	Defined by
`JMSExpiration`	Contains the message's absolute expiration time, expressed in milliseconds since January 1, 1970 UTC. WebLogic JMS determines expiration during send processing from the effective time-to-live, including any configured destination override. A `JMSExpiration` value of zero on a delivered message means that the message has no expiration time. Setting `JMSExpiration` directly on the message does not control its time-to-live. Expiration handling follows the destination's configured expiration policy: an expired message can be discarded, logged and discarded, or redirected to an error destination.	`send()` method
`JMSMessageID`	Contains a string value that uniquely identifies each message sent by a JMS Provider. This field is set internally by `send()`. All `JMSMessageID`s start with an `ID:` prefix. This value is overwritten by a call to `producer.send()`, setting this value directly on the message has no effect. The values set by the producer can be queried using the message supplied to `producer.send()` or when the message is received by a consumer. When the message is received, it contains a provider-assigned value.	`send()` method
`JMSPriority`	Specifies the priority level. This field is set on the producer or as parameter sent by the application before `send()`. JMS defines ten priority levels, 0 to 9, 0 being the lowest priority. Levels 0-4 indicate gradations of *normal* priority, and level 5-9 indicate gradations of *expedited* priority. When the message is received, it contains the value specified by the method sending the message.	`send()` method
`JMSRedelivered`	Specifies a flag set when a message is redelivered because no acknowledge was received. This flag is of interest to a receiving application. If set, the flag indicates that JMS may have delivered the message previously because one of the following is true: • The application has already received the message, but did not acknowledge it. • The session's `recover()` method was called to restart the session beginning after the last acknowledged message. For more information about the `recover()` method, see Recovering Received Messages.	WebLogic JMS
`JMSReplyTo`	Specifies a queue or topic to which reply messages should be sent. This field is set directly on the message by the application before `send()`. This feature can be used with the `JMSCorrelationID` header field to coordinate request/response messages. Setting the `JMSReplyTo` field does not guarantee a response; it simply *enables* the receiving application to respond.	Application
`JMSTimestamp`	Contains the time at which the message was sent. WebLogic JMS writes the timestamp in the message when it accepts the message for delivery, *not* when the application sends the message. When the message is received, it contains the timestamp. The value stored in the field is a Java millis time value.	WebLogic JMS

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

Table 1-9 (Cont.) Message Header Fields

Field	Description	Defined by
`JMSType`	Specifies the message type identifier (String) set directly on the message by the application before `send()`. The JMS specification allows some flexibility with this field to accommodate diverse JMS providers. Some messaging systems allow application-specific message types to be used. For such systems, the `JMSType` field could be used to hold a message type ID that provides access to the stored type definitions. WebLogic JMS does not restrict the use of this field.	Application

Message Property Fields

The property fields of a message contain header fields added by the sending application. The properties are standard Java name/value pairs. Property names must conform to the message selector syntax specifications defined in the `jakarta.jms.Message` Javadoc at `jakarta.jms.Message`. The following values are valid: boolean, byte, double, float, int, long, short, and String.

WebLogic Server supports the use of the following JMS (JMSX) defined properties as defined in the JMS Specification, at Jakarta Messaging 3.1 specification:

Table 1-10 JMSX Property

Type	Description
JMSXUserID	System generated property that identifies the user sending the message. See Using the JMSXUserID Property.
JMSXDeliveryCount	System generated property that specifies the number of message delivery attempts where first attempt is 1
JMSXGroupID	Identity of the message group
JMSXGroupSeq	Sequence number of a message within a group

Although message property fields may be used for application-specific purposes, JMS provides them primarily for use in message selectors. You determine how the JMS properties are used in your environment. You can include them in some messages and omit them from others depending upon your processing criteria. For more information, see:

- Setting and Browsing Message Header and Property Fields
- Filtering Messages
- JMS Specification, described at Jakarta Messaging 3.1 specification

Message Body

A message body contains the content being delivered from the producer to the consumer.

The following table summarizes standard JMS message types and the WebLogic `XMLMessage` extension. All message types extend `jakarta.jms.Message`, at `jakarta.jms.Message`, which consists of message headers and properties, but no message body.

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

Table 1-11 JMS Message Types

Type	Description
`jakarta.jms.BytesMessage`	Stream of uninterpreted bytes, which must be understood by the sender and receiver. The access methods for this message type are stream-oriented readers and writers based on `java.io.DataInputStream` and `java.io.DataOutputStream`. See `jakarta.jms.BytesMessage`.
`jakarta.jms.MapMessage`	A set of named values whose supported types include Java primitive types, `String`, and `byte[]`. Values are accessed by name; `getMapNames()` enumerates the names. See `jakarta.jms.MapMessage`.
`jakarta.jms.ObjectMessage`	Single serializable Java object. See `jakarta.jms.ObjectMessage`.
`jakarta.jms.StreamMessage`	A sequential stream of typed values, including Java primitive types, `String`, and `byte[]`, read and written through the corresponding typed methods. See `jakarta.jms.StreamMessage`.
`jakarta.jms.TextMessage`	Single String. The TextMessage can also contain XML content. See `jakarta.jms.TextMessage`.
`weblogic.jms.extensions.XMLMessage`	A WebLogic extension of `TextMessage` for XML content, with methods to set and retrieve the body as a DOM `Document`. See XMLMessage.

For more information, see the `jakarta.jms.Message` Javadoc at `jakarta.jms.Message`. For more information about the access methods and, if applicable, the conversion charts associated with a particular message type, see the Javadoc for that message type.

ServerSessionPoolFactory

> **Note**
>
> Session pool and connection consumer configuration objects are deprecated. They are not a required part of the Jakarta EE specification, do not support JTA user transactions, and are largely superseded by message driven beans (MDBs), which are simpler, easier to manage, and more capable. For more information about designing MDBs, see Message-Driven EJBs in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

A `jakarta.jms.ServerSessionPool` provides server sessions for concurrent message processing. WebLogic JMS supplies the `weblogic.jms.extensions.ServerSessionPoolFactory` extension to create a server-side pool.

WebLogic JMS binds each factory in JNDI under `weblogic.jms.ServerSessionPoolFactory:<jms-server-runtime-name>`, where the suffix is the name of the JMS server runtime instance. Applications retrieve the factory through JNDI and use it to create a server session pool.

For information about the WebLogic server session pool factory extension, see `weblogic.jms.extensions.ServerSessionPoolFactory` in Java API Reference for Oracle WebLogic Server.

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

ServerSessionPool

A `ServerSessionPool` application server object provides a pool of server sessions that connection consumers can retrieve in order to process messages concurrently.

A `ServerSessionPool` is created by the `ServerSessionPoolFactory` object (see ServerSessionPoolFactory) obtained through a JNDI lookup.

For information about the `ServerSessionPool` interface, see `jakarta.jms.ServerSessionPool` in the Jakarta Messaging 3.1 API documentation.

ServerSession

A `ServerSession` application server object enables you to associate a thread with a JMS session by providing a context for creating, sending, and receiving messages.

A `ServerSession` application is created by a `ServerSessionPool` object, described in ServerSessionPool.

For information about the `ServerSession` interface, see `jakarta.jms.ServerSession` in the Jakarta Messaging 3.1 API documentation.

ConnectionConsumer

A `ConnectionConsumer` object uses a server session to process received messages. If message traffic is heavy, then the connection consumer can load each server session with multiple messages to minimize thread context switching. A `ConnectionConsumer` is created by a Connection object, described in Connection.

For information about the `ConnectionConsumer` interface, see `jakarta.jms.ConnectionConsumer` in the Jakarta Messaging 3.1 API documentation.

> **Note**
>
> Connection consumer listeners run on the same JVM as the server.

Value-Added Public JMS API Extensions

WebLogic JMS is tightly integrated into the WebLogic Server platform, enabling you to build highly secure Jakarta EE applications that can be easily monitored and administered through the WebLogic Remote Console.

WebLogic JMS supports XA transactions and provides high-availability mechanisms through clustering and service migration. Cross-version communication and integration with third-party messaging providers are subject to the applicable compatibility requirements and supported integration mechanisms.

For a detailed listing of these value-added features, see WebLogic Server Value-Added JMS Features in *Administering JMS Resources for Oracle WebLogic Server*.

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

WebLogic Server Value-Added JMS Features

In addition to the standard JMS APIs specified by the JMS Specification, WebLogic Server provides numerous `weblogic.jms.extensions` APIs, which include the classes and methods described in the following table. For more information about these APIs, see *Java API Reference for Oracle WebLogic Server*.

Table 1-12 WebLogic JMS Public API Extensions

Interface/Class	Function
ConsumerInfo, DestinationInfo	Provides consumer and destination information to management clients in CompositeData format.
JMSMessageFactoryImpl, WLMessageFactory	Provides a factory and methods to: • Create JMS messages • Create JMS bytes messages • Create JMS map messages • Create JMS object messages • Create JMS stream messages • Create JMS text messages • Create JMS XML messages
JMSMessageInfo	Represents a JMS message and associated management metadata in the CompositeData format used by message-management APIs.
JMSModuleHelper, JMSNamedEntityModifier	JMSModuleHelper locates JMS runtime MBeans and manages JMS module configuration entities. JMSNamedEntityModifier supplies a callback for modifying a named configuration entity through JMSModuleHelper.
JMSRuntimeHelper	Monitors JMS runtime JMX MBeans
MDBTransaction	Associates a message delivered to a MDB (message-driven bean) with a transaction
WLDestination	Determines if a destination is a queue or a topic
WLMessage	Provides WebLogic-specific message information, including the Unit-of-Order name, distributed-destination forwarding status, and an XML DOM representation of the message.
WLMessageProducer	Configures producer extension options, including time-to-deliver, redelivery limits, send timeouts, and Unit-of-Order names.
WLJMSContext	Provides additional fields and methods that are not supported by `jakarta.jms.JMSContext`. WLJMSContext provides the same extension features as WLConnection and WLSession
WLJMSProducer	Provides additional methods that are not supported by `jakarta.jms.JMSProducer`.
WLQueueSession, WLSession, WLTopicSession	Provides additional fields and methods that are not supported by `jakarta.jms.QueueSession`, `jakarta.jms.Session`, and `jakarta.jms.TopicSession`
XMLMessage	Represents XML message content and provides DOM Document accessors.
Schedule	Computes the next time matching a schedule, or the interval until that time.

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

Table 1-12 (Cont.) WebLogic JMS Public API Extensions

Interface/Class	Function
JMSHelper	Legacy helper for JMS runtime and configuration MBeans and destination management. Deprecated since WebLogic Server 9.0; use JMSModuleHelper.
ServerSessionPoolFactory, ServerSessionPoolListener	Provides interfaces for creating server session pools and message listeners **Note:** Session pool configuration objects are deprecated. They are not a required part of the Jakarta EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are a required part of Jakarta EE. For more information on designing MDBs, see *Developing Message-Driven Beans for Oracle WebLogic Server*.

This API also supports `NO_ACKNOWLEDGE` and `MULTICAST_NO_ACKNOWLEDGE` acknowledge modes, and extended exceptions, including throwing an exception:

- To the session exception listener (if set), when one of its consumers has been closed by the server as a result of a server failure or administrative intervention.
- From a multicast session when the number of messages received by the session, but not yet delivered to the message listener, exceeds the maximum number of messages allowed for that session.
- From a multicast consumer when it detects a sequence gap (message received out of sequence) in the data stream.

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

2

Best Practices for Application Design

Learn about the design options for WebLogic Server JMS, application behaviors to be considered during the design process, and the recommended design patterns.

Message Design

Learn how to design messages to improve messaging performance.

Serializing Application Objects

The CPU cost of serializing Java objects can be significant. This expense, in turn, affects JMS Object messages. You can offset some of this cost by having application objects implement the `java.io.Externalizable` interface, but there still will be significant overhead in marshalling the class descriptor. To avoid the cost of having to write the class descriptors of additional objects embedded in an Object message, have these objects implement `Externalizable`, and call `readExternal` and `writeExternal` on them directly. For example, call `obj.writeExternal(stream)` rather than the `stream.writeObject(obj)`. Using `Bytes` and `Stream` messages is generally a preferred practice.

Serializing Strings

Serializing text can add CPU and memory overhead. A Java String's memory footprint depends on its content and the JVM's representation, while its serialized size depends on the encoding. In addition, the introduction of string-based messages often implies an expensive parse step in the application in order to process the String into something the application can make direct use of. Bytes, Stream, Map and Object messages are therefore sometimes preferable to Text and XML messages. Similarly, it is preferable to avoid the use of strings in message properties, especially if they are large.

Server-side Serialization

Delivery mode alone does not determine where serialization costs occur. Both clients and servers can perform serialization when messages cross JVM boundaries. The server also serializes message data for persistent storage and can serialize nonpersistent messages for paging. The cost depends on the message type, representation, and delivery path.

Selection

Selectors add evaluation work. Put data needed by standard JMS selectors in user-defined message properties or supported JMS header fields; standard selectors do not inspect the message body. Choose property types that match the selector expressions, and evaluate selector performance with representative message backlogs and consumer counts.

Message Compression

Compressing large messages in a JMS application can improve performance.

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

Compressing a message body can reduce the bytes transmitted and, when compressed data is retained, its memory or storage footprint. The benefit depends on the payload, compression method, and delivery path. Compression and decompression consume CPU on the clients or servers that perform them. WebLogic's automatic compression can run in a producer JVM or in a server JVM when messages are sent to a remote consumer.

Measure compression with representative messages and workloads. Small or already compressed payloads may provide little benefit or become larger after compression. For application-managed compression, Java provides the `java.util.zip` package.

For information about using JMS connection factories to specify the automatic compression of messages that exceed a specified threshold size, see Compressing Messages in the *Tuning Performance of Oracle WebLogic Server*.

Message Properties and Message Header Fields

Store application payload in the message body. Use small user-defined properties for selection or routing metadata, and use standard JMS header fields only for their documented purposes.

Large property values add memory, serialization, and copying costs. Paging can reduce memory use, but it does not eliminate all per-message memory overhead. Keep headers and properties small even when paging is enabled.

See Message Header Fields and Message Property Fields.

Message Ordering

Use Message Unit-of-Order when ordered processing is required for messages that share a Unit-of-Order within a destination. Applications must submit messages in the intended order; processing follows the feature's acknowledgement and transaction rules.

The advantages of Message Unit-of-Order over Ordered Redelivery are:

- Ease of configuration.
  - Does not require a custom connection factory for asynchronous receivers, such as setting the `MessagesMaximum` to 1 when using message-driven beans (MDBs).
  - Simple configuration when using distributed destinations.
- Preserves message order during processing delays.
- Preserves message order during transaction rollback or session recovery.

Oracle recommends applications that use Ordered Redelivery upgrade to Message Unit-of-Order. See Using the Message Unit-of-Order.

Topics Vs. Queues

Choose queues or topics according to whether consumers should share the work of receiving messages or receive independent copies through subscriptions.

A queue distributes deliveries among eligible consumers. Selection depends on message selectors, consumer capacity, and applicable ordering constraints; applications must not rely on a fixed round-robin order.

A topic delivers a copy to each eligible subscription. An unshared subscription can have at most one active consumer, while a shared subscription can distribute its messages among

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

multiple consumers. A durable subscription can retain messages while it is inactive; a nondurable subscription does not provide that offline-retention guarantee.

When the existing durable subscription can be removed, recreating it with the same identity but a different selector replaces the subscription and discards its retained messages. Close its consumers and complete outstanding transactions before attempting the change; the absence of active consumers alone does not establish that removal is permitted. Creating a queue consumer with a different selector does not delete unmatched messages from the queue.

For more information about configuring JMS queues and topics, see Queue and Topic Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

Asynchronous Vs. Synchronous Consumers

In general, asynchronous (`onMessage`) consumers perform and scale better than synchronous consumers.

- Asynchronous consumers create less network traffic. Messages are pushed unidirectionally, and are pipelined to the message listener. Pipelining supports the aggregation of multiple messages into a single network call.

  > **Note**
  >
  > – In WebLogic Server, your synchronous consumers can also use the same efficient behavior as asynchronous consumers by enabling the Prefetch Mode for Synchronous Consumers option on JMS connection factories, as described in Use Prefetch Mode to Create a Synchronous Message Pipeline.
  >
  > – WebLogic Server JMS does not support synchronous and asynchronous consumers on the same session.

- Asynchronous consumers use fewer threads. An asynchronous consumer does not use a thread while it is inactive. A synchronous consumer consumes a thread for the duration of its receive call. As a result, a thread can remain idle for long periods, especially if the call specifies a blocking timeout.
- For application code that runs on a server, it is almost always best to use asynchronous consumers, typically through MDBs. The use of asynchronous consumers prevents the application code from doing a blocking operation on the server. A blocking operation, in turn, idles a server-side thread; it can even cause deadlocks. Deadlocks occur when blocking operations consume all threads. When no threads remain to handle the operations required to unblock the blocking operation itself, that operation never stops blocking.

For more information, see Receiving Messages Asynchronously using the Classic API and Receiving Messages Synchronously Using the Classic API.

Persistent Vs. Non Persistent Messages

Choose the delivery mode according to the application's failure-recovery requirements.

Use persistent delivery when messages need the provider-failure protection defined by the JMS contract. Nonpersistent delivery can reduce storage I/O but can lose messages on provider failure. Evaluate performance together with the required recovery guarantees; persistent-store write policies also affect those guarantees.

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

Determine the effective delivery mode as follows:

1. For the classic API, a newly created `MessageProducer` receives its initial delivery mode from the connection factory. The default connection-factory value is `Persistent`. Calling `MessageProducer.setDeliveryMode()` changes the producer's setting. A delivery-mode argument on an individual `send()` or legacy `publish()` call overrides that setting for the call.
2. For the simplified API, a newly created `JMSProducer` uses `DeliveryMode.PERSISTENT`. Use `JMSProducer.setDeliveryMode()` to change its setting before sending.
3. The destination's delivery-mode override can force incoming messages to `Persistent` or `Non-Persistent`. The default, `No-Delivery`, leaves the producer's choice unchanged.
4. When the JMS server's `StoreEnabled` setting is true, it uses its configured persistent store or, where supported, the hosting WebLogic Server instance's default store. When `StoreEnabled` is false, persistent sends are rejected unless `AllowsPersistentDowngrade` is true. If downgrading is allowed, the effective delivery mode becomes nonpersistent.

The `DeliveryMode` interface defines `PERSISTENT` and `NON_PERSISTENT`. The constant `Message.DEFAULT_DELIVERY_MODE` equals `DeliveryMode.PERSISTENT`; it does not mean “inherit the connection-factory setting.”

Temporary destinations and nondurable topic subscriptions do not retain messages durably across a JMS server restart. This storage behavior does not itself change the message's `JMSDeliveryMode` header. Topic messages are retained for eligible subscriptions according to their lifetime, acknowledgement, transaction, expiration, and delivery-mode rules.

To retain durable-subscription metadata across JMS server restarts, keep persistence enabled on the JMS server and use a supported persistent-store configuration, even when the subscription receives only nonpersistent messages.

> **Note**
>
> Do not infer message delivery mode from persistent-store file growth alone. A store also contains durable-subscription metadata and other persistent service state, and its file size can reflect allocation and reuse.

Deferring Acknowledges and Commits

Processing messages in bounded batches before acknowledging or committing can reduce completion overhead when the application can tolerate batch redelivery after a failure.

Use `CLIENT_ACKNOWLEDGE` for explicit acknowledgement of non-transacted receives. With the default `AcknowledgePolicy` of `All`, acknowledging one message acknowledges all unacknowledged messages consumed by that session. Ensure that all messages covered by the acknowledgement have been processed successfully. For a locally transacted session, use `Session.commit()` to complete the batch; rollback can cause its received messages to be delivered again. When JTA or a container controls the transaction, use that transaction's completion rules.

Bound each batch by both message count and elapsed time so that a final partial batch can complete. A synchronous consumer on a dedicated session can use a finite receive timeout to detect that no further message has arrived before the batch deadline. Complete the batch on the thread that owns the session, and measure the throughput, latency, memory and redelivery tradeoffs with representative traffic.

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

Do not use synchronous receives, including `receiveNoWait()`, on a session with registered message listeners. Do not have a timer thread acknowledge or commit a listener's session. For listener-based processing, select an acknowledgement or transaction boundary that completes successfully processed work without depending on another message arriving.

Acknowledgement Optimization for Nondurable Subscribers

WebLogic JMS can reduce acknowledgement traffic for eligible non-transacted, nondurable topic consumers by maintaining recovery information and handling acknowledgements on the client.

This optimization depends on the consumer and destination settings; it is not used for every nondurable subscription. Redelivery delay, redelivery limits and shared-subscription configuration can affect eligibility. The optimization does not change the application's acknowledgement-mode contract.

Measure throughput and client memory use with representative message sizes, subscriber counts and processing rates. Retained message data and buffered deliveries contribute to heap use and garbage-collection work. Size the client heap and message buffers from those measurements, and verify the result under sustained load and slow-consumer conditions.

Alternative Qualities of Service, Multicast and No-Acknowledge

WebLogic JMS provides alternative qualities of service (QOS) extensions that can help performance.

Using MULTICAST_NO_ACKNOWLEDGE

`WLSession.MULTICAST_NO_ACKNOWLEDGE` is a WebLogic acknowledgement mode for non-transacted, nondurable subscribers to a topic configured for multicast. Receive these messages asynchronously with a message listener; synchronous receives are not supported in this mode. Multicast can reduce the number of message copies sent by the server when many subscribers receive the same data, but performance and scalability depend on the network and subscriber workload.

> **Note**
>
> Each multicast session schedules a long-running client socket reader. Account for this thread demand, together with message-listener work, when sizing the JMS client's execution resources.

Multicast delivery can lose messages because of network congestion or client overrun, and duplicate delivery is possible. `Session.recover()` cannot recover multicast messages, and `Message.acknowledge()` does not acknowledge them. Use this mode only when the application can tolerate missing and duplicate messages.

Using NO_ACKNOWLEDGE

`WLSession.NO_ACKNOWLEDGE` is a WebLogic acknowledgement mode for non-transacted sessions. The server does not retain delivered messages for later client acknowledgement, and `Session.recover()` cannot recover them. This can reduce acknowledgement traffic, but messages can be lost or delivered more than once if delivery fails.

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

> **Note**
>
> Closing an asynchronous consumer can discard unprocessed messages already in its client pipeline. In this mode, the server does not retain those messages for redelivery.

A smaller connection-factory `MessagesMaximum` value can limit the number of buffered messages exposed to loss when a consumer closes or a failure occurs. Measure the throughput tradeoff; reducing the buffer does not make this acknowledgement mode reliable.

Use Sessions from One Thread at a Time

Use a separate `Session` for each thread that performs messaging concurrently. A session and its producers and consumers must be used serially.

Once a connection is started, a session with registered message listeners is controlled by the message-delivery thread, including between callbacks. The permitted concurrent `Session.close()` and `MessageConsumer.close()` operations are exceptions. Adding producers to the same session does not provide additional concurrent execution. Avoid concurrent modification of a `Message`, and do not reuse a message before an asynchronous send completes. See Threading restrictions on a session and Asynchronous send in the Jakarta Messaging 3.1 specification.

Using the JMSXUserID Property

For ordinary send operations, WebLogic JMS can attach the authenticated sending user's name as the `JMSXUserID` message property. The connection factory's `AttachJMSXUserId` setting requests this behavior; the destination's `AttachSender` setting determines whether to support that request, always attach the name, or never attach it.

Consider the following points when using the `JMSXUserID` property in your application.

- `JMSXUserID` is an optional JMS-defined property. Its population and forwarding rules are provider-specific; do not assume that other providers use the WebLogic rules described here.
- For an ordinary send, WebLogic JMS derives the value from the authenticated subject used for that send operation. Depending on the client's security configuration, this can be the calling thread's subject or a subject associated with the connection.
- For an ordinary send, WebLogic JMS clears or replaces an application-supplied `JMSXUserID`. Setting the message property directly does not select the authenticated identity that WebLogic attaches.
- `JMSXUserID` is message metadata, not a cryptographic signature. A TLS-enabled transport such as `t3s` protects data on that transport connection; it does not establish end-to-end message-origin verification across forwarding intermediaries.
- WebLogic JMS Store-and-Forward forwarding clears the source message's `JMSXUserID` before forwarding it.
- WebLogic Messaging Bridge does not preserve the original sender's identity in `JMSXUserID`, even when Preserve Message Properties is enabled. Depending on the bridge and target configuration, `JMSXUserID` can be absent or identify the bridge's authenticated sending user. This applies to both same-domain and cross-domain configurations.

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

- The WebLogic `WLMessageProducer.forward()` extension can preserve a received message's `JMSXUserID`, subject to the target destination's `AttachSender` policy. A target configured with `AttachSender=Never` removes the value.

> **Note**
>
> The `JMSXUserID` property interoperability behavior for WebLogic JMS clients prior to 9.0 is undetermined.

Performance and Tuning

Implement the performance tuning features available with WebLogic JMS and get the most out of your applications.

See Tuning WebLogic JMS in *Tuning Performance of Oracle WebLogic Server*.

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

3

Enhanced Support for Using WebLogic JMS with EJBs and Servlets

Learn how to use WebLogic JMS wrappers when accessing WebLogic JMS or third-party JMS providers from EJBs and servlets. The wrappers provide resource pooling and container integration for JMS operations.

Enabling WebLogic JMS Wrappers

WebLogic Server uses JMS wrappers that make it easier to use WebLogic JMS inside a Jakarta EE component, such as an EJB or a servlet.

The JMS wrappers also provide a number of enhanced usability and performance features:

- Automatic pooling of JMS connection and session objects (and some pooling of message producer objects as well)
- Automatic transaction enlistment for WebLogic JMS implementations and for third-party JMS providers that support two-phase commit transactions (XA protocol)
- Testing of the JMS connection, as well as reestablishment after a failure
- Security credentials that are managed by the EJB or servlet container

The following sections provide information on how to use WebLogic JMS wrappers:

Declaring a JMSContext Object Using @Inject Annotation

In a Jakarta EE component that supports CDI injection, use @Inject to obtain a container-managed JMSContext. The following example selects a JMS connection factory:

```
@Inject
@JMSConnectionFactory("myJMSCF")
private JMSContext context;
```

The container manages the injected JMSContext. Its underlying context is created when first used in the active transaction or request and is closed when that scope ends. Application code must not close an injected JMSContext.

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

> **Note**
>
> - Injection should be enabled for the class. Depending on the class being used and the archive in which it is packaged, it may be necessary to specify a `beans.xml` file. For more information, see Using Contexts and Dependency Injection for the Jakarta EE Platform in *Developing Applications for Oracle WebLogic Server*.
> - If the field is null, verify that the containing object is managed by the container and that CDI injection is enabled for its class and archive. Check the application's CDI configuration, including beans.xml when required. If using a non-null injected JMSContext fails, inspect the exception and server log, and verify the connection-factory lookup name and availability.

Specifying a Lookup Name in JMSContext Injection

When injecting a `JMSContext` object, you can use the `@JMSConnectionFactory` annotation to specify the product-specific global JNDI look up name of a connection factory to be used by the container.

> **Note**
>
> When you provide a product-specific global JNDI name for the connection factory annotation, you cannot override it using a resource reference in the deployment descriptor of the container.

Alternatively, you can specify a fully qualified resource reference name of the form `java:comp/env/res-ref-name` as follows:

```
@Inject
@JMSConnectionFactory("java:comp/env/res-ref-name")
private JMSContext context;
```

In this case, declare the component's connection-factory resource reference and map it to the desired connection factory. You can declare the reference with a resource-ref element in the deployment descriptor or with an appropriate @Resource declaration. See Declaring a Wrapped JMS Factory using Deployment Descriptors.

If @JMSConnectionFactory is omitted, the container uses the Jakarta EE default JMS connection factory at java:comp/DefaultJMSConnectionFactory. When the annotation is present, supply its connection-factory lookup name.

Determining the Authentication Type for JMSContext Injection

When `@JMSConnectionFactory` names a connection-factory resource reference, the reference's authentication setting applies. To supply a user name and password through `@JMSPasswordCredential`, configure the resource reference with `res-auth` set to `Application`. WebLogic Server rejects explicit credentials when the reference uses `Container` authentication.

Omitting `@JMSPasswordCredential` does not change a resource reference's authentication setting. A reference configured for `Container` continues to use container authentication. A

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

reference configured for `Application` creates the context without explicitly supplied credentials; the JMS provider determines whether that connection can be authenticated.

For a direct connection-factory lookup handled by the WebLogic JMS wrappers, the credentials in `@JMSPasswordCredential` are used when the annotation is present. Container authentication is used when the annotation is absent.

Declaring JMS Objects as Resources In the EJB or Servlet Deployment Descriptors

The following sections provide information on declaring JMS objects as resources:

For more information about packaging EJBs, see Implementing Enterprise JavaBeans in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*. For more information about programming servlets, see Creating and Configuring Servlets in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Declaring a Wrapped JMS Factory using Deployment Descriptors

> **Note**
>
> New applications will likely use EJB 3.0 annotations instead of deployment descriptors. Annotations are described in Declaring JMS Destinations and Connection Factories Using Annotations.

You can declare a JMS connection factory as part of an EJB or servlet by defining a `resource-ref` element in the `ejb-jar.xml` or `web.xml` file, respectively. This process creates a "wrapped" JMS connection factory that can benefit from the more advanced session pooling, automatic transaction enlistment, connection monitoring, and container-managed security features described in Improving Performance Through Pooling.

Here is an example of such a connection factory element:

```
<resource-ref>
 <res-ref-name>jms/QCF</res-ref-name>
 <res-type>jakarta.jms.QueueConnectionFactory</res-type>
 <res-auth>Container</res-auth>
 <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

This element declares that a JMS `QueueConnectionFactory` object be bound into JNDI, at the location:

```
java:comp/env/jms/QCF
```

This JNDI name is only valid inside the context of the EJB or servlet where the `resource-ref` is declared, which is what the `java:comp/env` JNDI context signifies.

For the mapping shown here, add a matching resource-description element to weblogic-ejb-jar.xml for an EJB or to weblogic.xml for a web application. The res-ref-name must match the reference declared in the standard deployment descriptor. For example:

```
<resource-description>
 <res-ref-name>jms/QCF</res-ref-name>
```


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

```
 <jndi-name>weblogic.jms.ConnectionFactory</jndi-name>
</resource-description>
```

The example maps the reference to weblogic.jms.ConnectionFactory, one of the WebLogic default connection factories. This factory is available when JMS Default Connection Factories Enabled is enabled on the hosting WebLogic Server instance. To use another locally bound factory, specify its JNDI name in jndi-name. A Foreign Server mapping can provide a local JNDI name for a factory in another WebLogic domain or supplied by another JMS provider.

An invalid connection-factory mapping can fail when the reference is resolved or used. This may occur during component initialization or later during application execution; successful deployment does not by itself verify that the connection factory is available.

Declaring JMS Destinations using Deployment Descriptors

You can use a `jms-destination` element in a web, EJB, application-client, or application deployment descriptor to define a JMS destination resource. Use a `resource-env-ref` element to declare a reference to a destination; the reference does not create the destination.

> **Note**
>
> New applications will likely use EJB 3.2 annotations instead of deployment descriptors. Annotations are described in Declaring JMS Destinations and Connection Factories Using Annotations.

The transaction enlistment, pooling, connection monitoring features take place in the connection factory, not in the destinations. However, this feature is useful for consistency, and to make an application less dependent on a particular configuration of WebLogic Server, since destinations can easily be modified by simply changing the corresponding `jms-destination` or `resource-env-ref` description, without having to recompile the source code

Declaring JMS Destinations Using the jms-destination Element

You can define a JMS destination resource using the `jms-destination` element in the `ejb-jar.xml` or `web.xml` deployment descriptors. It creates the destination and binds it to the appropriate naming context based on the namespace specified.

The following example defines a queue destination `myQueue1` that is bound to JNDI at the location `java:app/MyJMSDestination`:

```
<jms-destination>
  <description>JMS Destination definition</description>
  <name>java:app/MyJMSDestination</name>
  <interface-name>jakarta.jms.Queue</interface-name>
  <destination-name>myQueue1</destination-name>
</jms-destination>
```

For the jms-destination element and its child elements, see jms-destinationType in the Jakarta EE 11 common schema.

Declaring JMS Destinations Using the resource-env-ref Element

You can also bind a JMS queue or topic destination into the `java:comp/env` JNDI tree by declaring it as a `resource-env-ref` element in the `ejb-jar.xml` or `web.xml` deployment descriptors.

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

A resource-env-ref declares a reference to a destination; it does not create that destination. The destination must be available when the reference is resolved. An invalid destination mapping can fail during injection or startup, or later when the application looks up or uses the destination.

Here is an example of such a queue destination element:

```
<resource-env-ref>
 <resource-env-ref-name>jms/TESTQUEUE</resource-env-ref-name>
 <resource-env-ref-type>jakarta.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

This element declares that a JMS `Queue` destination object will be bound into JNDI, at the location:

```
java:comp/env/jms/TESTQUEUE
```

As with a referenced connection factory, this JNDI name is only valid inside the context of the EJB or servlet where the `resource-ref` is declared.

You must also define a matching `resource-env-description` element in the `weblogic-ejb-jar.xml` or `weblogic.xml` file. This provides a layer of indirection that enables you to easily modify referenced destinations just by changing the corresponding `resource-env-ref` deployment descriptors.

```
<resource-env-description>
 <resource-env-ref-name>jms/TESTQUEUE</resource-env-ref-name>
 <jndi-name>jmstest.destinations.TESTQUEUE</jndi-name>
</resource-env-description>
```

Referencing a Packaged JMS Application Module In Deployment Descriptor Files

When you package a JMS module with an enterprise application, you must reference the JMS resources within the module in all applicable descriptor files of the Jakarta EE application components, including:

- The WebLogic enterprise descriptor file, `weblogic-application.xml`
- Any WebLogic deployment descriptor file, such as `weblogic-ejb-jar.xml` or `weblogic.xml`
- Any Jakarta EE descriptor file, such as EJB (`ejb-jar.xml`) or WebApp (`web.xml`) files

Referencing Application Modules in a weblogic-application.xml Descriptor

When including JMS modules in an enterprise application, you must list each JMS module as a module element of type JMS in the `weblogic-application.xml` descriptor file packaged with the application, and a path that is relative to the root of the Jakarta EE application. Here is an example of a reference to a JMS module name *Workflows*:

```
<module>
  <name>Workflows</name>
  <type>JMS</type>
  <path>jms/Workflows-jms.xml</path>
</module>
```


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

Referencing JMS Resources in a WebLogic Application

Within any `weblogic-foo` descriptor file, such as EJB (`weblogic-ejb-jar.xml`) or WebApp (`weblogic.xml`), the name of the JMS module is followed by a number (#) separator character, which is followed by the name of the resource inside the module. For example, a JMS module named *Workflows* that contains a queue named *OrderQueue*, would have a name of *Workflows#OrderQueue*.

```
<resource-env-description>
  <resource-env-ref-name>jms/OrderQueue</resource-env-ref-name>
  <resource-link>Workflows#OrderQueue</resource-link>
</resource-env-description>
```

Note that the `<resource-link>` element is unique to WebLogic Server, and is how the resources that are defined in a JMS module are referenced (linked) from the other Jakarta EE Application components.

Referencing JMS Resources in a Jakarta EE Application

Declare a connection-factory reference with resource-ref and a destination reference with resource-env-ref in the component's standard deployment descriptor. These reference names are the names used by the component in java:comp/env.

Match each res-ref-name to the res-ref-name in its WebLogic resource-description mapping. Match each resource-env-ref-name to the resource-env-ref-name in its WebLogic resource-env-description mapping. The mapping's resource-link identifies the JMS module and resource; the component reference name does not have to equal the JMS resource's name. For example, jms/OrderQueue can map to Workflows#OrderQueue, as shown in Referencing JMS Resources in a WebLogic Application.

The following declarations provide the component reference names and types:

```
<resource-ref>
  <res-ref-name>jms/OrderQueueFactory</res-ref-name>
  <res-type>jakarta.jms.ConnectionFactory</res-type>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>jms/OrderQueue</resource-env-ref-name>
  <resource-env-ref-type>jakarta.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

Declaring JMS Destinations and Connection Factories Using Annotations

WebLogic Server 10.0 and later releases support the EJB 3.0 programming model which uses annotations to configure metadata, eliminating the need for deployment descriptors. You can declare JMS objects using the @Resources annotation as described in Standard JDK Annotations Used By EJB 3.0 in *Developing Jakarta Enterprise Beans for Oracle WebLogic Server*.

Injecting Resource Dependency into a Class

Apply @Resource to a field or setter method to request injection of a JMS destination or connection factory into a container-managed component. The following example uses field injection. A class-level @Resource or @Resources declaration creates resource references for lookup; it does not inject a field. See Non-Injected EJB 3.0 Resource Reference Annotations.

Example 3-1 is a Wrapped JMS Pooling Annotation example:

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

Example 3-1 Wrapped JMS Pooling Annotation Example

```
.
.
.
// The "name=" or "type=" are not always required,
// "mappedName=" is usually sufficient.
@Resource(name="ReplyQueue",
 type=jakarta.jms.Queue.class,
 mappedName="jms/ReplyQueue") Destination rq;
.
.
.
@Resource(name="ReplyConnectionFactory",
 type=jakarta.jms.ConnectionFactory.class,
 mappedName = "jms/ConnectionFactory") ConnectionFactory cf;
.
.
.
```

Non-Injected EJB 3.0 Resource Reference Annotations

Injected resource dependencies are resolved when the host EJB or servlet is instantiated. You may not want injected resource because:

- The injection may prevent applications from deploying successfully if the container attempts to resolve references during deployment.
- You might want to defer reference resolution until the application is first invoked.

You can setup a non-injected resource reference by placing the `@Resources` annotation above the class definition. An application can resolve such references at runtime by looking up the reference in the bean context. As a best practice, the bean or servlet should also cache the result in order to avoid the overhead of repeated lookups as shown in Example 3-2:

For a full example, see EJB 3.0 Wrapper Without Injection.

Example 3-2 Non-Injected Resource Example

```
.
.
.
@Resources ({
  @Resource(name="targetCFRef",
            mappedName="TargetCFJNDIName",
            type=jakarta.jms.ConnectionFactory.class),

  @Resource(name="targetDestRef",
            mappedName="TargetDestJNDIName",
            type=jakarta.jms.Destination.class)
})

@Stateless(mappedName="StatelessBean")
public class MyStatelessBean implements MyStateless {

  @Resource
  private SessionContext sctx;   // inject the bean context

  private ConnectionFactory targetCF;
  private Destination targetDest;

  public void completeWorkOrder() {
```


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

```
        // Lookup the JMS resources and cache for re-use.  Note that a
        // "java:comp/env" prefix isn't needed for EJB3.0 bean contexts.

        if (targetCF == null) targetCF =
          (jakarta.jms.ConnectionFactory)sctx.lookup("targetCFRef");

        if (targetDest == null) targetDest =
          (jakarta.jms.Destination)sctx.lookup("targetDestRef");
.
.
.
```

Avoid Transactional XA Interfaces

With resource wrapping, do not use the `jakarta.jms` XA interfaces. The container uses them internally if the JMS code is used inside a transaction context. This allows your EJB application code to run EJBs in an environment where transactions are present or in a non-transactional environment, just by changing the deployment descriptors.

Disabling Wrapping and Pooling

It is sometimes desirable to leverage resource references but disable resource reference wrapping and pooling.

To disable resource wrapping and pooling, use the deployment descriptor approach, but change the `res-type` to `java.lang.Object` in the `resource-ref` stanza for the connection factory. There is currently no known way to disable wrapping and pooling using annotations.

What's Happening Under the JMS Wrapper Covers

Understand what is actually taking place under the covers when WebLogic Server creates a set of wrappers around the JMS objects.

For example, the code fragment in Sending a JMS Message in a Jakarta EE Container, shows an instance of a WebLogic-specific wrapper class being returned, rather than the actual JMS connection factory because the connection factory was looked up from the `java:comp/env` JNDI tree. This wrapper object intercepts certain calls to the JMS provider and inserts the correct Jakarta EE behavior, as described in the following sections.

Automatically Enlisting Transactions

A wrapped JMS session can participate automatically in an active Jakarta transaction when the selected connection factory provides the required transaction support. This applies to container-managed EJB transactions and to transactions demarcated with UserTransaction where application-managed transaction demarcation is permitted.

If the wrapped session cannot participate in the active transaction, the classic JMS operation fails with a JMSException. The current message identifier is 169822:

```
Unable to use a wrapped JMS session in the transaction because two-phase commit is not
available.
```

Use a connection factory configured for XA participation when the messaging operation must be part of the Jakarta transaction. Otherwise, arrange for that operation to run outside an active transaction. For an EJB using container-managed transactions, use the NOT_SUPPORTED transaction attribute (NotSupported in a deployment descriptor) where

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

appropriate. In a servlet or bean-managed-transaction EJB, use the permitted UserTransaction boundaries to control whether the messaging operation is transactional.

Container-Managed Security

WebLogic JMS uses the security credentials that are present on the thread when the EJB or servlet container is invoked. For foreign JMS providers, however, when you declare a JMS connection factory through a `resource-ref` element in the `ejb-jar.xml` or `web.xml` file, there is an optional sub element called `res-auth`. This element may have one of two settings:

Container — When you set the `res-auth` element to `Container`, security to the JMS provider is managed by the Jakarta EE container. In this case, if the JMS connection factory was mapped into the JNDI tree using a Foreign JMS Connection Factory configuration MBean, then the user name and password from that MBean is used. Otherwise, WebLogic Server connects to the provider with no user name or password specified and throws an error if the `createConnection()` method is used to pass a user name and password to the connection factory.

Application — When you set the `res-auth` element to `Application`, any user name or password on the MBean is ignored. Instead, the application code must specify a user name and password to the `createConnection(String userName, String password)` method of the JMS connection factory, or use the version of `createConnection()` with no parameters if the user name or password are not required.

> **Note**
>
> Injecting a `JMSContext` does not override a connection-factory resource reference's authentication setting. To supply credentials through `@JMSPasswordCredential` when using such a reference, configure it for `Application` authentication. See Determining the Authentication Type for JMSContext Injection.

Connection Testing

The JMS wrappers register an ExceptionListener on pooled connections to detect provider-reported failures. When connection health checking is enabled and a temporary destination and consumer can be created, a pool tester sends and receives a test message at two-minute intervals.

If health checking is disabled or the temporary-message test cannot be initialized, those probes are not run; exception-listener monitoring remains available. For foreign connection factories, ConnectionHealthChecking controls the temporary-message test and defaults to enabled.

Jakarta EE Compliance

EJB and servlet applications must follow Jakarta Messaging restrictions. For the classic API, do not call:

- On Connection: createConnectionConsumer, createDurableConnectionConsumer, createSharedConnectionConsumer, createSharedDurableConnectionConsumer, setClientID, setExceptionListener, or stop.
- On Session: getMessageListener, setMessageListener, or run.

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

- On MessageConsumer: getMessageListener or setMessageListener.

These containers also restrict JMSContext methods, limit each Connection to one active Session, and prohibit CompletionListener-based asynchronous sends. See sections 12.2 and 12.3 of the Jakarta Messaging 3.1 specification for the complete rules.

For portable code using the classic API, create sessions with createSession(false, Session.AUTO_ACKNOWLEDGE). An active JTA transaction controls completion of participating JMS operations. When a Session is created through the classic API without an active JTA transaction, nontransacted sessions using AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE retain their specified behavior; support for local transactions and CLIENT_ACKNOWLEDGE is optional and should not be assumed in portable applications. See Automatically Enlisting Transactions.

> **Note**
>
> Use a message-driven bean for asynchronous message receipt in an EJB application. See *Developing Message-Driven Beans for Oracle WebLogic Server*.

Pooled JMS Connection Objects

The JMS wrappers pool various session objects in order to make code like the example provided in Sending a JMS Message in a Jakarta EE Container more efficient. A pooled JMS connection is a session pool used by EJBs and servlets that use a `resource-ref` element in their deployment descriptor to define their JMS connection factories, as discussed in Declaring a Wrapped JMS Factory using Deployment Descriptors.

Improving Performance Through Pooling

The automatic pooling of connections and other objects by the JMS wrappers means that it is efficient to write code.

For example, see the example in Sending a JMS Message in a Jakarta EE Container. Although in this example the Connection Factory, Connection, and Session objects are created every time a message is sent, in reality these three classes work together so that when they are used as shown, they do little more than retrieve a Session object from the pool.

Speeding Up JNDI Lookups by Pooling Session Objects

The JNDI lookups of the Connection Factory and Destination objects can be expensive in terms of performance. This is particularly true if the Destination object points to a Foreign JMS Destination MBean, and therefore, is a lookup on a non local JNDI provider. Because the Connection Factory and Destination objects are thread-safe, they can be looked up after they are inside an EJB or servlet at creation time, which saves the time required to perform the lookup each time.

Inside a servlet, these lookups can be performed inside the `init()` method. The Connection Factory and Destination objects can then be assigned to an instance variable and reused whenever a message is sent.

Inside an EJB, these lookups can be performed inside the `ejbCreate()` method and assigned to an instance variable. For a session bean, each instance of the bean will then have its own copy. Because stateless session beans are pooled, this method is also very efficient (and is consistent with the Jakarta EE specifications), because the number of a times that lookups

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

occur is drastically reduced by pooling the JMS connection objects. (Caching these objects in a static member of the EJB class may work, but it is discouraged by the Jakarta EE specification.)

Cached connection factories and destinations may become unusable after a provider failure. If an operation fails because a cached object is no longer valid, invalidate that object and repeat its JNDI lookup before a later attempt. Recovery depends on the provider and configuration; a server restart does not always require a new lookup. PoolTestBean.java illustrates invalidating cached references after a failure and looking them up again on a later invocation.

Speeding Up Object Creation Through Caching

Acquire connections, sessions, and producers when needed and close them after use so the wrappers can return their underlying resources to the pool. Keeping these handles for the lifetime of a bean reduces the resources available for reuse. After a provider failure, discard failed handles and acquire new ones. Do not rely on pool recovery to repair a handle already returned to the application.

Enlisting the Proper Transaction Mode

Use an XA-capable connection factory when JMS work must participate in JTA. A container-managed EJB transaction or a transaction begun through UserTransaction can provide that context. Servlets may use UserTransaction; they do not use EJB container-managed transaction attributes. The wrappers enlist a foreign XA resource as needed, while WebLogic JMS uses its native transaction integration. An operation cannot participate in JTA if the required transaction support is unavailable.

Outside an existing JTA transaction, wrapper transaction handling depends on the provider and operation. An XA session can require a transaction around an individual operation, but native WebLogic JMS producer sends are not automatically wrapped in a new JTA transaction solely because the factory supports XA.

Choose a connection factory according to the required transaction semantics, and measure overhead with a representative workload. Do not disable XA for a resource used by work that must participate in JTA.

Simplified Access to Foreign JMS Providers

Learn how to access foreign JMS providers by using WebLogic Remote Console.

Foreign JMS server configuration maps connection factories and destinations from an external JNDI provider into the local WebLogic JNDI tree. The provider can be another WebLogic Server cluster or domain, or a third-party JMS provider. See Accessing Foreign JMS Providers in *Administering JMS Resources for Oracle WebLogic Server*.

The configuration contains the following resources:

- Foreign server: Specifies the remote JNDI provider's initial context factory, connection URL, and additional JNDI properties. It contains the foreign connection-factory and destination mappings and determines where those mappings are deployed. Target the foreign server to the applicable WebLogic server or cluster. See ForeignServerBean in the *MBean Reference for Oracle WebLogic Server*.
- Foreign connection factory: Maps a local JNDI name to the connection factory's JNDI name in the remote provider. It can also specify a user name and password for container-managed connection authentication. A non-replicated local binding is created on each

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

WebLogic Server instance where its parent foreign server is deployed. See ForeignConnectionFactoryBean in the *MBean Reference for Oracle WebLogic Server*.

- Foreign destination: Maps a local JNDI name to a destination's JNDI name in the remote provider.

Examples of JMS Wrapper Functions

JMS wrapper functions make it easier to use WebLogic JMS inside a Jakarta EE component, such as an EJB or a servlet.

Stateless Session Bean Example

The following files define a stateless session bean with an EJB2.x remote/home view. Its sendXATransactional method sends a message through JMS resource references within the container-managed transaction configured by the deployment descriptors. Configure the TESTQUEUE destination before deployment. The later MDB example shows a separate message-driven bean that uses JMS resource references.

ejb-jar.xml

This section describes the EJB components. For the "JMS wrapper" code examples provided in this section, note that this section declares the `resource-ref` and `resource-env-ref` elements for the wrapped JMS connection factory (`QueueConnectionFactory`) and referenced JMS destination (`TESTQUEUE`).

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/
j2ee/ejb-jar_2_1.xsd"
         version="2.1">
 <enterprise-beans>
 <session>
  <ejb-name>PoolTestBean</ejb-name>
  <home>weblogic.jms.pool.test.PoolTestHome</home>
  <remote>weblogic.jms.pool.test.PoolTest</remote>
  <ejb-class>weblogic.jms.pool.test.PoolTestBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>

  <resource-ref>
   <res-ref-name>jms/QCF</res-ref-name>
   <res-type>jakarta.jms.QueueConnectionFactory</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>

  <resource-env-ref>
  <resource-env-ref-name>jms/TESTQUEUE</resource-env-ref-name>
   <resource-env-ref-type>jakarta.jms.Queue</resource-env-ref-type>
  </resource-env-ref>
 </session>
 </enterprise-beans>

 <assembly-descriptor>
  <container-transaction>
   <method>
    <ejb-name>PoolTestBean</ejb-name>
```


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

```
    <method-name>*</method-name>
   </method>
   <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

weblogic-ejb-jar.xml

The resource-description element maps the jms/QCF connection-factory reference to weblogic.jms.XAConnectionFactory. The resource-env-description element maps the jms/TESTQUEUE destination reference to TESTQUEUE. Configure that destination before deploying the application.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-ejb-jar xmlns="http://www.bea.com/ns/weblogic/920"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://www.bea.com/ns/weblogic/920 http://
www.bea.com/ns/weblogic/920/weblogic-ejb-jar.xsd">
 <weblogic-enterprise-bean>
  <ejb-name>PoolTestBean</ejb-name>
  <stateless-session-descriptor>
   <pool>
    <max-beans-in-free-pool>8</max-beans-in-free-pool>
    <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
   </pool>
  </stateless-session-descriptor>

  <resource-description>
   <res-ref-name>jms/QCF</res-ref-name>
   <jndi-name>weblogic.jms.XAConnectionFactory</jndi-name>
  </resource-description>
  <resource-env-description>
   <resource-env-ref-name>jms/TESTQUEUE</resource-env-ref-name>
   <jndi-name>TESTQUEUE</jndi-name>
  </resource-env-description>
  <jndi-name>PoolTest</jndi-name>
 </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

PoolTest.java

This section defines the "remote" interface for the `PoolTest` bean. It declares one method, called `sendXATransactional`.

```
package weblogic.jms.pool.test;

import java.rmi.*;
import jakarta.ejb.*;
public interface PoolTest extends EJBObject
{
 public String sendXATransactional(String text)
  throws RemoteException;
}
```

PoolTestHome.java

This section defines the home interface required by this example's EJB2.x remote client view.

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

```
package weblogic.jms.pool.test;

import java.rmi.*;
import jakarta.ejb.*;

public interface PoolTestHome
 extends EJBHome
{
 PoolTest create()
  throws CreateException, RemoteException;
}
```

PoolTestBean.java

This section defines the actual EJB code. It sends a message whenever the `sendXATransactional` method is called.

```
package weblogic.jms.pool.test;

import java.lang.reflect.*;
import java.rmi.*;
import jakarta.ejb.*;
import jakarta.jms.*;
import javax.naming.*;
import jakarta.transaction.*;

public class PoolTestBean
 implements SessionBean
{
 private SessionContext context;
 private QueueConnectionFactory qcf;
 private Queue destination;

 public void ejbActivate()
 {
 }

 public void ejbRemove()
 {
 }

 public void ejbPassivate()
 {
 }

 public void setSessionContext(SessionContext ctx)
 {
  context = ctx;
 }

 private void lookupJNDIObjects()
  throws NamingException
 {
  InitialContext ic = new InitialContext();
  try {
   qcf =
    (QueueConnectionFactory)ic.lookup
                    ("java:comp/env/jms/QCF");
   destination =
    (Queue)ic.lookup("java:comp/env/jms/TESTQUEUE");
   } finally {
    ic.close();
```


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

```
  }
 }

 public void ejbCreate()
  throws CreateException
 {
  try {
   lookupJNDIObjects();
  } catch (NamingException ne) {
   throw new CreateException(ne.toString());
  }
 }

 public String sendXATransactional(String text)
  throws RemoteException
 {
  String id = "Not sent yet";
  try {
   if ((qcf == null) || (destination == null)) {
    lookupJNDIObjects();
   }
   QueueConnection connection = qcf.createQueueConnection();
   try {
    QueueSession session = connection.createQueueSession
                           (false, Session.AUTO_ACKNOWLEDGE);
    TextMessage message = session.createTextMessage
                           (text);
    QueueSender sender = session.createSender(destination);
    sender.send(message);
    id = message.getJMSMessageID();
   } finally {
    connection.close();
   }
  } catch (Exception e) {
   // Invalidate the JNDI objects if there is a failure.
   // A later invocation will look them up again.
   // Recovery behavior depends on the JMS provider.
   qcf = null;
   destination = null;
   throw new RemoteException("Failure in EJB: " + e);
  }
  return id;
 }
}
```

Sending a JMS Message in a Jakarta EE Container

After you declare the JMS connection factory and destination resources, you can use them to send and receive JMS messages inside an EJB or servlet. The following sections provide examples of how to send a message:

Using comp/env

The code in Example 3-3 sends a message if you map to the `java:comp/env` JNDI tree:

Example 3-3 Sending a Message Using comp/env

```
QueueConnectionFactory qcf;
Queue destQueue;
InitialContext ic = new InitialContext();
try {
```


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

```
 qcf = (QueueConnectionFactory)ic.lookup("java:comp/env/jms/QCF");
 destQueue = (Queue)ic.lookup("java:comp/env/jms/TESTQUEUE");
} finally {
 ic.close();
}
QueueConnection connection = qcf.createQueueConnection();
try {
 QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
 QueueSender sender = session.createSender(destQueue);
 TextMessage msg = session.createTextMessage("This is a test");
 sender.send(msg);
} finally {
 connection.close();
}
```

This is standard code that complies with the Jakarta EE specification and should run on any EJB or servlet product that properly supports Jakarta EE , the difference is that it runs more efficiently on WebLogic Server, because under the covers various objects are pooled, as described in Pooled JMS Connection Objects.

Note that this code example uses a `try...finally` block to guarantee that the `close()` method on the JMS Connection object is executed even if one of the statements inside the block throws an exception. If no connection pooling were being done, then this block would be necessary in order to ensure that the connection is closed, and to prevent server resources from being wasted. But because WebLogic Server pools some of the objects that are created by this code example , it is even more important that `close()` be called; otherwise, the EJB or servlet container will not know when to return the object to the pool.

Also, none of the transactional XA extensions to the JMS API are used in this code example . Instead, the container uses them internally if the JMS code is used inside a transaction context. But whether or not XA is used internally, the user-written code is the same, and does not use any JMS XA classes. This is what is specified by Jakarta EE. Writing EJB code in this way enables you to run EJBs in an environment where transactions are present or in a non-transactional environment, just by changing the deployment descriptors.

> **Note**
>
> When an EJB uses a wrapped JMS connection factory obtained through a resource reference in java:comp/env, application code must not use the jakarta.jms XA interfaces. The container manages transaction integration.

Dependency Injection

The code in Example 3-4 sends a message if you have used dependency injection to a variable.

`MyStateless.java`

```
package test;

public interface MyStateless {
    void completeWorkOrder();
}
```


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

Example 3-4 Sending a Message using Dependency Injection

```
package test;
// Example injected annotation.
import jakarta.annotation.Resource;
import jakarta.ejb.*;
import jakarta.jms.*;

@Stateless(mappedName="StatelessBean")
public class MyStatelessBean implements MyStateless {
  @Resource(mappedName="myDestJNDIName")
  private Destination dest;

 @Resource(mappedName="weblogic.jms.XAConnectionFactory")
  private ConnectionFactory connectionFactory;

  public void completeWorkOrder() {
    Connection con =  null;
    Session session = null;
    MessageProducer sender = null;
    try {
      System.out.println("completeWorkOrder called!");
      con = connectionFactory.createConnection();
      session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
      sender = session.createProducer(null);
      Message message = session.createTextMessage("work order complete!");
      sender.send(dest, message);
    } catch(Exception e) {
      throw new EJBException("Exception sending message:   " + e, e);
    } finally {
      try {
        if (con != null) con.close();
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

EJB 3.0 Wrapper Without Injection

Example 3-5 demonstrates EJB 3.0 annotations for an MDB that references resources that are not injected. The references are resolved at runtime when the MDB is invoked instead of when the MDB instances are instantiated.

Example 3-5 Non injected MDB Example

```
package test;

import jakarta.annotation.Resources;
import jakarta.annotation.Resource;
import javax.naming.*;
import jakarta.ejb.*;
import jakarta.jms.*;

import jakarta.ejb.ActivationConfigProperty;

@MessageDriven(
     name = "MyMDB",
     mappedName = "JNDINameOfMDBSourceDest",
     activationConfig = {
```


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

```
     // JMS interface type of the MDB destination:
     // jakarta.jms.Topic or jakarta.jms.Queue
     @ActivationConfigProperty(
          propertyName = "destinationType",
          propertyValue = "jakarta.jms.Queue"),
     // optionally specify a connection factory
     // there's no need to specify a connection factory if the source
     // destination is a WebLogic JMS destination
     @ActivationConfigProperty(
          propertyName = "connectionFactoryJndiName",
          propertyValue = "JNDINameOfMDBSourceCF"),
    })

// resources that are not injected

@Resources ({
  @Resource(name="targetCFRef",
            mappedName="TargetCFJNDIName",
            type=jakarta.jms.ConnectionFactory.class),

  @Resource(name="targetDestRef",
            mappedName="TargetDestJNDIName",
            type=jakarta.jms.Destination.class)
})


public class MyMDB implements MessageListener {

  // inject a reference to the MDB context

  @Resource
  private MessageDrivenContext mdctx;

  // cache targetCF and targetDest for re-use (performance)

  private ConnectionFactory targetCF;
  private Destination targetDest;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void onMessage(Message message) {

    Connection jmsConnection = null;

    try {
      System.out.println("My MDB got message: " + message);

      if (targetCF == null)
        targetCF = (jakarta.jms.ConnectionFactory)mdctx.lookup("targetCFRef");

      if (targetDest == null)
        targetDest = (jakarta.jms.Destination)mdctx.lookup("targetDestRef");

      jmsConnection = targetCF.createConnection();
      Session s = jmsConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageProducer mp = s.createProducer(null);

      if (message.getJMSReplyTo() != null)
        mp.send(message.getJMSReplyTo(), s.createTextMessage("My Reply"));
      else
        mp.send(targetDest, message);

    } catch (JMSException e) {
```


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

```
        throw new EJBException(e);

      } finally {

        // Return JMS resources to the resource reference pool for later re-use.
        // Closing a connection automatically also closes its sessions, etc.

        try { if (jmsConnection != null) jmsConnection.close(); }
        catch (JMSException ignored) {};
      }
    }
  }
```


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

4

Understanding the Simplified API Programming Model

Understand the key features of JMS simplified API defined by the Java Message Service (JMS) 2.0 specification. Also learn how it is implemented for creating JMS applications for WebLogic Server.

About JMS 2.0 Simplified API

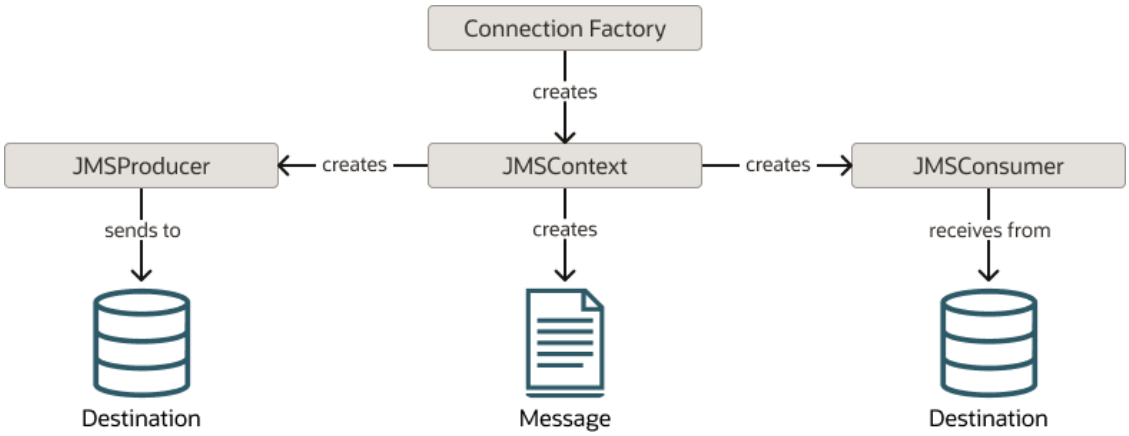
The JMS 2.0 simplified API provides the same basic functionality as the JMS 1.1 API (classic API), but the new interfaces and several API changes make it easier to use.

The following interfaces provided by the simplified API were implemented in Oracle WebLogic Server 12.2.1 release:

- `ConnectionFactory`: An administered object used to create a `JMSContext` through its `createContext` methods in the simplified API, or a `Connection` through its `createConnection` methods in the classic API.
- `JMSContext` : An active connection to a JMS provider and a single-threaded context used to send or receive messages.
- `JMSProducer` : An object created by a `JMSContext` to send messages to a queue or topic.
- `JMSConsumer` : An object created by a `JMSContext` to receive messages sent to a queue or topic.

Figure 4-1 shows how these objects fit together in a JMS client application.

Figure 4-1 Simplified API Programming Model



For more information, see the jakarta.jms package in the Jakarta Messaging 3.1 API.

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

New Interfaces in the Simplified JMS API

The JMS 2.0 simplified API consists of three new interfaces.

JMSContext

JMSContext is the main interface in the simplified API. It combines the connection and session functionality that the classic API exposes through separate Connection and Session objects. To send text without explicitly creating a TextMessage, use the JMSProducer.send(Destination, String) overload, which creates the text message for the supplied body. Use JMSContext.createTextMessage when the application needs a TextMessage object.

For more information, see the JMSContext interface in the Jakarta Messaging 3.1 API.

The `WLJMSContext` interface in the `weblogic.jms.extensions` package defines the fields and methods that are not supported by `jakarta.jms.JMSContext`. It provides the same extension features as `WLConnection` and `WLSession`. See the Javadoc for `WLJMSContext` in *Java API Reference for Oracle WebLogic Server*.

JMSProducer

To send messages in the simplified API, use a `JMSProducer` object. You can create a `JMSProducer` object by calling the `createProducer` method on a `JMSContext` object.

> **Note**
>
> You do not need to save the `JMSProducer` object in a variable. It is recommended that you create this object when sending a message. For more information, see Sending Messages Using the Simplified JMS API.

For more information, see the JMSProducer interface in the Jakarta Messaging 3.1 API.

The WLJMSProducer interface in weblogic.jms.extensions provides WebLogic JMS extensions, including setUnitOfOrder and setRedeliveryLimit. To use these extensions, obtain a WebLogic JMS producer and cast it to WLJMSProducer. JMSProducer implementations supplied by other providers are not required to implement this interface. See the Javadoc for WLJMSProducer in *Java API Reference for Oracle WebLogic Server*.

JMSConsumer

The `JMSConsumer` object receives messages from a queue or topic. You can create a `JMSConsumer` object by passing a `Queue` or `Topic` object to one of the `createConsumer` methods on a `JMSContext` or by passing a `Topic` object to one of the `createSharedConsumer` or `createDurableConsumer` methods on a `JMSContext` object.

For more information, see the JMSConsumer interface in the Jakarta Messaging 3.1 API.

New Methods to Simplify Messaging in JMS 2.0

In addition to the methods for sending and receiving messages on `JMSContext` objects, JMS 2.0 introduces a few more methods to simplify the code.

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

Method to Extract the Body Directly from a Message

The `getBody` method provides an easy way to obtain the body from a message. This method applies to both the classic and simplified API.

The following statement assumes that message is a jakarta.jms.Message referring to a received BytesMessage. The enclosing method must catch or declare jakarta.jms.JMSException. If the statement is used in MessageListener.onMessage, catch JMSException inside the callback; onMessage cannot declare it.

```
byte[] bytes = message.getBody(byte[].class);
```

For more information, see the Message interface in the Jakarta Messaging 3.1 API.

Method to Receive a Message Body Directly

The `receiveBody` method can be used to receive any type of message except for `StreamMessage` and `Message`, as long as the class of the expected body is known in advance.

```
JMSConsumer consumer = ...
String body = consumer.receiveBody(String.class,1000);
```

For more information, see the JMSConsumer interface in the Jakarta Messaging 3.1 API.

Method to Create a Session

A new `createSession` method, that accepts a single parameter or no parameter, was added to the `jakarta.jms.Connection`. See Create a Session Using the createSession Method.

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

5

Developing a Basic JMS Application

Learn how to set up a basic WebLogic JMS application using the JMS 2.0 and JMS 1.1 APIs.

Importing Required Packages

Import the Java packages that define all required classes and interfaces to create, send, receive, and read messages for the WebLogic application.

Table 5-1 lists the packages that are commonly used by WebLogic JMS applications.

Table 5-1 WebLogic JMS Packages

Package	Description
`jakarta.jms`	JMS API. This package is always used by WebLogic JMS applications. See `jakarta.jms package`.
`javax.naming` weblogic.jndi	JNDI packages required for server and destination lookups. See `Java SE 21 JNDI API`.
`jakarta.transaction.UserTransaction`	JTA API required for JTA user transaction support. See `https://jakarta.ee/specifications/transactions/` and `https://jakarta.ee/specifications/transactions/2.0/apidocs/`.
weblogic.jms.extensions	WebLogic-specific JMS public API that provides additional classes and methods, as described in Value-Added Public JMS API Extensions.

Setting Up a JMS Application

Before you can send and receive messages, you must set up a JMS application.

The following sections describe the procedure to set up a basic WebLogic JMS application:

You must ensure that the system administrator responsible for configuring WebLogic Server has configured the required JMS resources, including the connection factories, JMS servers, and destinations.

- For information about JMS resource definitions, see Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*.
- For information about configuring other JMS resources, see Messaging in the *Oracle WebLogic Remote Console Online Help.*
- For more information about the JMS interfaces and methods described in these sections, see Understanding the JMS API, or the `jakarta.jms`, at `jakarta.jms package`, or the weblogic.jms.extensions Javadoc in *Java API Reference for Oracle WebLogic Server*.
- For information about setting up transacted applications and JTA user transactions, see Using Transactions with WebLogic JMS.

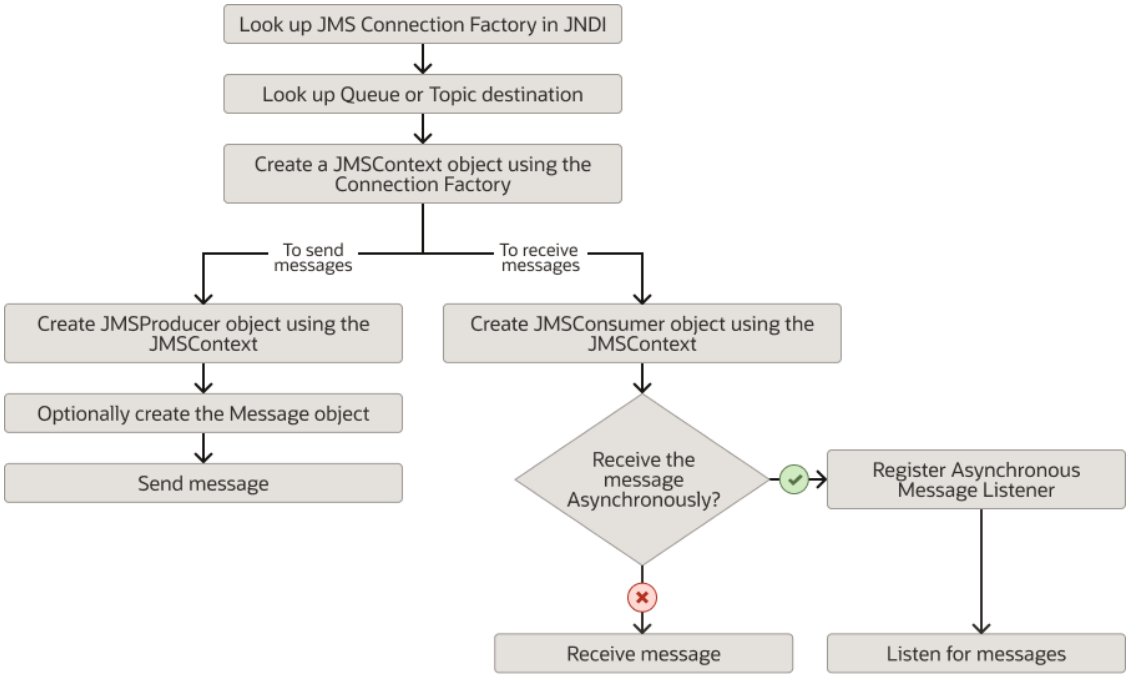
Copyright © 2007, 2026, Oracle and/or its affiliates.

Using a Simplified API to Set Up a JMS Application

Oracle WebLogic Server 12.2.1 supports the JMS 2.0 simplified API for sending and receiving messages. For more information about the simplified API, see Understanding the Simplified API Programming Model .

Figure 5-1 shows the steps required to set up a JMS application using the JMS 2.0 Simplified API.

Figure 5-1 Setting Up a JMS Application Using the Simplified API



Look Up a Connection Factory in JNDI

Before you can look up a connection factory, it must be defined as part of the configuration information.

Use a unique configured name for each connection factory within a JMS module, and configure the JNDI name that clients use to look it up. You can also use the default connection factories defined by the Jakarta EE specification and WebLogic Server. For information, see Connection Factory Configuration in *Administering JMS Resources for Oracle WebLogic Server*.

After the connection factory is defined, establish the JNDI context used below with the `InitialContext` constructor. Code running in WebLogic Server, including servlets and EJBs, can use `new InitialContext()` for the local naming service. For a remote lookup, configure the appropriate JNDI environment properties, including the initial context factory and provider URL; you can pass them in a `Hashtable` to `new InitialContext(environment)`. For constructor details, see `InitialContext`.

After the JNDI context is defined, to look up a connection factory in JNDI, execute the following command:

```
ConnectionFactory connectionFactory =
 (ConnectionFactory) namingContext.lookup(CF_name);
```


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

The `CF_name` argument specifies the connection factory's JNDI lookup name. It need not equal the connection factory's configured Name.

For more information about the `ConnectionFactory` interface, see ConnectionFactory, or the `jakarta.jms.ConnectionFactory` Javadoc, at `ConnectionFactory`.

Look Up a Queue or Topic

Before you can look up a queue or a topic, it must be configured by the WebLogic JMS system administrator, as described in Create a JMS System Module and Configure Resources for JMS System Modules in the *Oracle WebLogic Remote Console Online Help.* For more information, see Destination or the Javadocs at `Queue` and `Topic`.

After the destination is configured, you can look up a queue or topic destination using one of the following procedures:

You can look up a queue or topic destination by establishing a JNDI context (`namingContext`), which has already been accomplished in Look Up a Connection Factory in JNDI, and executing one of the following commands, for Point-to-Point or Publish/Subscribe messaging, respectively:

```
Queue queue = (Queue) namingContext.lookup(Queue_name);

Topic topic = (Topic) namingContext.lookup(Topic_name);
```

The `Queue_name` and `Topic_name` arguments specify the JNDI names of the queue and topic destinations defined during the configuration.

Create a JMSContext Object

A `JMSContext` object replaces the `Connection` and `Session` objects in the classic API. For more information, see New Interfaces in the Simplified JMS API.

The JMSContext object can be created by calling one of the several `createContext` methods on a `ConnectionFactory` object. For example:

```
JMSContext context = connectionFactory.createContext(sessionMode);
```

In this case, a connection and a session with the specified mode are created for use by the new `JMSContext` object `context`. For more information, see `connectionFactory` interface definition in `ConnectionFactory`.

Alternatively, you can inject `JMSContext` in the Jakarta EE web and EJB containers using the `@Inject` annotation as described in Declaring a JMSContext Object Using @Inject Annotation. This is the recommended way for creating `JMSContext` in Jakarta EE applications. For example:

```
@Inject @JMSConnectionFactory("myJMSCF") JMSContext context;
```

For more information about using the `JMSContext` interface, see `JMSContext`.

Create JMSProducer and JMSConsumer Objects

Use the JMSProducer and JMSConsumer objects to send and receive messages respectively.

You can create a `JMSProducer` object by calling the `createProducer` method on a `JMSContext` object as follows:

```
JMSProducer producer = context.createProducer();
```


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

> **Note**
>
> You do not need to save the `JMSProducer` object in a variable. Instead, create the object while calling the `send` method as follows:
>
> ```
> context.createProducer().send(queue, message);
> ```

For more information, see `JMSProducer`.

You can create a `JMSConsumer` object by passing a queue or topic object to one of the `createConsumer` methods on a `JMSContext` object as follows:

```
JMSContext context = connectionFactory.createContext();
JMSConsumer consumer = context.createConsumer(queue);
```

For more information, see `JMSConsumer`.

Sending and Receiving Messages using the Simplified API

The following sections describe how to send and receive messages using the Simplified API:

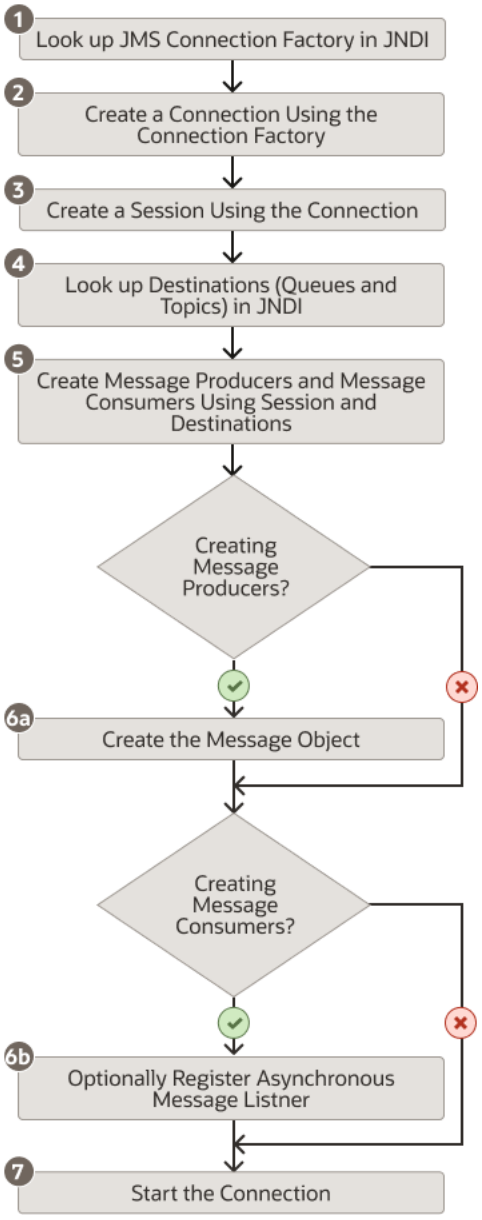
- Sending Messages Using the Simplified JMS API
- Sending a Message Asynchronously
- Receive Messages Asynchronously Using the Simplified API
- Receive Messages Synchronously Using the Simplified API

Using the Classic API to Set Up a JMS Application

The following figure shows the steps required to set up a JMS application using JMS 1.1 classic API.

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

Figure 5-2 Setting Up a JMS Application using Classic API



> **Note**
>
> Oracle WebLogic Server 12.2.1 supports JMS 2.0 simplified API for sending and receiving messages. See Understanding the Simplified API Programming Model .

Step 1: Look Up a Connection Factory in JNDI

Before you can look up a connection factory, it must be defined as part of the configuration information.

Use a unique configured name for each connection factory within a JMS module, and configure the JNDI name that clients use to look it up. You can also use the default connection factories

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

defined by the Jakarta EE specification and WebLogic Server. For information, see "Connection Factory Configuration" in *Administering JMS Resources for Oracle WebLogic Server*.

After the connection factory is defined, establish the JNDI context used below with the `InitialContext` constructor. Code running in WebLogic Server, including servlets and EJBs, can use `new InitialContext()` for the local naming service. For a remote lookup, configure the appropriate JNDI environment properties, including the initial context factory and provider URL; you can pass them in a `Hashtable` to `new InitialContext(environment)`. For constructor details, see `InitialContext`.

After the context is defined, to look up a connection factory in JNDI, execute one of the following commands, for PTP or Publish/Subscribe messaging, respectively:

```
QueueConnectionFactory queueConnectionFactory =
 (QueueConnectionFactory) context.lookup(CF_name);

TopicConnectionFactory topicConnectionFactory =
 (TopicConnectionFactory) context.lookup(CF_name);
```

The `CF_name` argument specifies the connection factory's JNDI lookup name. It need not equal the connection factory's configured Name.

For more information about the `ConnectionFactory` interface, see ConnectionFactory, or the `jakarta.jms.ConnectionFactory` Javadoc, at `ConnectionFactory`.

Step 2: Create a Connection Using the Connection Factory

You can create a connection for accessing the messaging system by using the `ConnectionFactory` methods described in the following sections.

For more information about the `Connection` interface, see Connection, or the `jakarta.jms.Connection` Javadoc, at `Connection`.

Create a Queue Connection

The `QueueConnectionFactory` provides the following two methods for creating a queue connection:

```
public QueueConnection createQueueConnection(
) throws JMSException

public QueueConnection createQueueConnection(
 String userName,
 String password
) throws JMSException
```

The first method creates a `QueueConnection`; the second method creates a `QueueConnection` using a specified user identity. In each case, a connection is created in stopped mode and must be started in order to accept messages, as described in Step 7: Start the Connection.

For more information about the `QueueConnectionFactory` interface methods, see the `jakarta.jms.QueueConnectionFactory` Javadoc, at `QueueConnectionFactory`. For more information about the `QueueConnection` interface, see the `jakarta.jms.QueueConnection` Javadoc, at `QueueConnection`.

Create a Topic Connection

The `TopicConnectionFactory` provides the following two methods to create a topic connection:

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

```
public TopicConnection createTopicConnection(
) throws JMSException

public TopicConnection createTopicConnection(
 String userName,
 String password
) throws JMSException
```

The first method creates a `TopicConnection`; the second method creates a `TopicConnection` using a specified user identity. In each case, a connection is created in stopped mode and must be started in order to accept messages, as described in Step 7: Start the Connection.

For more information about the `TopicConnectionFactory` interface methods, see the `jakarta.jms.TopicConnectionFactory` Javadoc., at `TopicConnectionFactory`. For more information about the `TopicConnection` interface, see the `jakarta.jms.TopicConnection` Javadoc, at `TopicConnection`.

Step 3: Create a Session Using the Connection

You can create one or more sessions for accessing a queue or topic using the `Connection` methods described in the following sections.

> **Note**
>
> A generic `Session` can create producers and consumers for both queues and topics, subject to the restrictions of the destination and acknowledgement mode. The domain-specific `QueueSession` and `TopicSession` interfaces retain their queue-only and topic-only restrictions.
>
> Use a separate session for each thread that performs messaging concurrently. Once the connection is started, a session with registered message listeners and its producers and consumers must not be used by other application threads; this restriction also applies between callbacks. The permitted concurrent `Session.close()` and `MessageConsumer.close()` operations are exceptions. See the `jakarta.jms.Session` API documentation for the complete threading rules.

For more information about the `Session` interface, see Session or the `jakarta.jms.Session` Javadoc, at `Session`.

Create a Session Using the createSession Method

Use the `createSession` method in `jakarta.jms.Connection` to create a session. This method accepts a single parameter, `sessionMode`, or no parameter as follows:

```
Session createSession(int sessionMode) throws JMSException

or

Session createSession() throws JMSException
```

Create a Queue Session

The `QueueConnection` class defines the following method for creating a queue session:

```
public QueueSession createQueueSession(
 boolean transacted,
```


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

```
 int acknowledgeMode
) throws JMSException
```

You must specify a boolean argument indicating whether the session will be transacted (`true`) or non-transacted (`false`), and an integer that indicates the acknowledge mode for non-transacted sessions. The `acknowledgeMode` attribute is ignored for transacted sessions. In this case, messages are acknowledged when the transaction is committed using the `commit()` method.

For more information about the `QueueConnection` interface methods, see the `jakarta.jms.QueueConnection` Javadoc, at `QueueConnection`. For more information about the `QueueSession` interface, see the `jakarta.jms.QueueSession` Javadoc, at `QueueSession`.

Create a Topic Session

The `TopicConnection` class defines the following method for creating a topic session:

```
public TopicSession createTopicSession(
 boolean transacted,
 int acknowledgeMode
) throws JMSException
```

You must specify a boolean argument indicating whether the session will be transacted (`true`) or non-transacted (`false`), and an integer that indicates the acknowledge mode for non-transacted sessions. The `acknowledgeMode` attribute is ignored for transacted sessions. In this case, messages are acknowledged when the transaction is committed using the `commit()` method.

For more information about the `TopicConnection` interface methods, see the `jakarta.jms.TopicConnection` Javadoc, at `TopicConnection`. For more information about the `TopicSession` interface, see the `jakarta.jms.TopicSession` Javadoc, at `TopicSession`.

Step 4: Look Up a Destination (Queue or Topic)

Before you can look up a destination, the destination must be configured by the WebLogic JMS system administrator, as described in Create a JMS System Module and Configure Resources for JMS System Modules in the *Oracle WebLogic Remote Console Online Help.* For more information about the `Destination` interface, see Destination or the `jakarta.jms.Destination` Javadoc, at `Destination`.

After the destination is configured, you can look up a destination using a JNDI name or a reference:

Using a JNDI Name

You can look up a destination by establishing a JNDI context (`context`), which has already been accomplished in Look Up a Connection Factory in JNDI, and executing one of the following commands, for PTP or Publish/Subscribe messaging, respectively:

```
Queue queue = (Queue) context.lookup(Dest_name);

Topic topic = (Topic) context.lookup(Dest_name);
```

The *`Dest_name`* argument specifies the JNDI name of the destination defined during configuration.

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

Use a Reference

If you do not use a JNDI namespace, you can use the following `QueueSession` or `TopicSession` method to reference a queue or topic, respectively:

> **Note**
>
> The `createQueue()` and `createTopic()` methods *do not* create destinations dynamically; they create only references to destinations that already exist. For information about creating destinations dynamically, see Using JMS Module Helper to Manage Applications.

```
public Queue createQueue(
 String queueName
) throws JMSException

public Topic createTopic(
 String topicName
) throws JMSException
```

For the syntax of JNDI name, createQueue(), and createTopic(), see How to Look Up a Destination.

Step 5: Create Message Producers and Message Consumers

You can create message producers and message consumers by passing the destination reference to the `Session` methods described in the following sections.

> **Note**
>
> Each consumer receives its own local copy of a message. After a message is received, you can modify the header field values; however, the message properties and message body are read only. (Attempting to modify the message properties or body at this point will generate a `MessageNotWriteableException`.) You can modify the message body by executing the corresponding message type's `clearBody()` method to clear the existing contents and enable the write permission.

For more information about the `MessageProducer` and `MessageConsumer` interfaces, see MessageProducer and MessageConsumer, or the `jakarta.jms.MessageProducer`, at `MessageProducer`, and `jakarta.jms.MessageConsumer` Javadocs, at `MessageConsumer`.

Create QueueSenders and QueueReceivers

The `QueueSession` object defines the following methods for creating queue senders and receivers:

```
public QueueSender createSender(
 Queue queue
) throws JMSException

public QueueReceiver createReceiver(
```


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

```
 Queue queue
) throws JMSException

public QueueReceiver createReceiver(
 Queue queue,
 String messageSelector
) throws JMSException
```

You must specify the queue object for the queue sender or receiver being created. You may also specify a message selector for filtering messages. Message selectors are described in more detail in Filtering Messages.

If you pass `null` to `createSender()`, you create an anonymous producer. Specify a `Queue` object when sending each message, as described in Sending Messages.

Use `getQueue()` to obtain the `Queue` object associated with a `QueueSender` or `QueueReceiver`. An anonymous `QueueSender` returns `null`:

```
public Queue getQueue(
) throws JMSException
```

For more information about the `QueueSession` interface methods, see the `jakarta.jms.QueueSession` Javadoc, at `QueueSession`. For more information about the `QueueSender` and `QueueReceiver` interfaces, see the `jakarta.jms.QueueSender`, at `QueueSender`, and `jakarta.jms.QueueReceiver` Javadocs, at `QueueReceiver`.

Create TopicPublishers and TopicSubscribers

The `TopicSession` object defines the following methods for creating topic publishers and topic subscribers:

```
public TopicPublisher createPublisher(
 Topic topic
) throws JMSException

public TopicSubscriber createSubscriber(
 Topic topic
) throws JMSException

public TopicSubscriber createSubscriber(
 Topic topic,
 String messageSelector,
 boolean noLocal
) throws JMSException
```

> **Note**
>
> The `createSubscriber` methods shown here create non-durable subscribers, whose subscriptions last only while those subscribers exist. A durable subscription can retain eligible messages while it has no active consumer. Those messages remain available until acknowledged or removed, for example by expiration. A consumer that reconnects to the same durable subscription can receive its retained messages; creating a new subscription does not retrieve earlier publications. See Creating Subscribers for a Durable Subscription.

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

You must specify the topic object for the publisher or subscriber being created. You can specify a message selector for filtering messages and a `noLocal` flag (described later in this section). Message selectors are described in more detail in Filtering Messages.

If you pass `null` to `createPublisher()`, you create an anonymous producer. Specify a `Topic` object when publishing each message, as described in Sending Messages.

An application can publish and subscribe using the same connection and can receive its own publications. To exclude messages published by that connection, set `noLocal` to `true` when creating the topic subscriber.

Use `getTopic()` to obtain the `Topic` object associated with a `TopicPublisher` or `TopicSubscriber`. An anonymous `TopicPublisher` returns `null`:

```
Topic getTopic(
) throws JMSException
```

In addition, you can access the `noLocal` variable setting associated with the topic subscriber using the following `TopicSubscriber` method:

```
boolean getNoLocal(
) throws JMSException
```

For more information about the `TopicSession` interface methods, see the `jakarta.jms.TopicSession` Javadoc, at `TopicSession`. For more information about the `TopicPublisher` and `TopicSubscriber` interfaces, see the `jakarta.jms.TopicPublisher`, at `TopicPublisher`, and the `jakarta.jms.TopicSubscriber` Javadocs, at `TopicSubscriber`.

Step 6a: Create the Message Object (Message Producers)

> **Note**
>
> This step applies to message producers only.

To create the message object, use one of the following `Session` or `WLSession` class methods:

- `Session` Methods

  > **Note**
  >
  > These methods are inherited by both the `QueueSession` and `TopicSession` subclasses.

```
public BytesMessage createBytesMessage(
) throws JMSException

public MapMessage createMapMessage(
) throws JMSException

public Message createMessage(
) throws JMSException

public ObjectMessage createObjectMessage(
) throws JMSException
```


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

```
public ObjectMessage createObjectMessage(
 Serializable object
) throws JMSException

public StreamMessage createStreamMessage(
) throws JMSException

public TextMessage createTextMessage(
) throws JMSException

public TextMessage createTextMessage(
 String text
) throws JMSException
```

- `WLSession` Method

```
public XMLMessage createXMLMessage(
 String text
) throws JMSException
```

For more information about the `Session` and `WLSession` interface methods, see the `jakarta.jms.Session`, at `Session`, and `weblogic.jms.extensions.WLSession` Javadocs, respectively. For more information about the `Message` interface and its methods, see Messages, or the `jakarta.jms.Message` Javadoc, at `Message`.

Step 6b: Optionally Register an Asynchronous Message Listener

> **Note**
>
> This step applies to message consumers in Java SE applications and the Jakarta EE application client container. Web and EJB applications must use message-driven beans for asynchronous message receipt; they must not register a MessageListener on a consumer. See Jakarta EE Compliance.

To receive messages asynchronously, you must register an asynchronous message listener by performing the following steps:

1. Implement the `jakarta.jms.MessageListener` interface, at `MessageListener`, which includes an `onMessage()` method.

> **Note**
>
> For an example of the `onMessage()` method interface, see Example: Setting Up a Point-to-Point JMS Application Using the Classic API.
>
> A message listener may close its own `MessageConsumer` from `onMessage()` even when Allow Close In OnMessage is `false`. Closing the listener's own `Session` or `Connection`, or stopping its own `Connection`, from the callback requires this WebLogic option to be `true`. The option defaults to `true` for custom connection factories; the default connection factories set it to `false` and do not allow it to be changed. For more information on configuring connection factory options, see Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*.

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

2. Set the message listener using the following `MessageConsumer` method, passing the listener information as an argument:

```
public void setMessageListener(
 MessageListener listener
) throws JMSException
```

3. Optionally, implement an exception listener on the session to catch exceptions, as described in Defining a Connection Exception Listener.

To remove the message listener, call `setMessageListener(null)` on the `MessageConsumer`.

After a message listener is defined, you can access it by calling the following `MessageConsumer` method:

```
public MessageListener getMessageListener(
) throws JMSException
```

> **Note**
>
> WebLogic JMS guarantees that multiple `onMessage()` calls for the same session will not be executed simultaneously.

If a message consumer is closed by an administrator or as the result of a server failure, then a `ConsumerClosedException` is delivered to the session exception listener, if one was defined. In this way, a new message consumer can be created, if necessary. For information about defining a session exception listener, see Defining a Connection Exception Listener.

The `MessageConsumer` interface methods are inherited by the `QueueReceiver` and `TopicSubscriber` interfaces. For additional information about the `MessageConsumer` interface methods, see MessageProducer and MessageConsumer or the `jakarta.jms.MessageConsumer` Javadoc, at `MessageConsumer`.

Step 7: Start the Connection

You start the connection using the `Connection` class `start()` method.

For additional information about starting, stopping, and closing a connection, see Starting, Stopping, and Closing a Connection or the `jakarta.jms.Connection` Javadoc, at `Connection`.

Example: Setting Up a Point-to-Point JMS Application Using the Classic API

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/queue`, where `WL_HOME` is the WebLogic Server installation directory. The `init()` method shows you how to set up and start a `QueueSession` for a JMS application. The following shows the `init()` method, with comments describing each setup step.

Define the required variables, including the JNDI context, JMS connection factory, and queue static variables.

```
public final static String JNDI_FACTORY=
       "weblogic.jndi.WLInitialContextFactory";
public final static String JMS_FACTORY=
       "weblogic.examples.jms.QueueConnectionFactory";
public final static String
```


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

```
        QUEUE="weblogic.examples.jms.exampleQueue";

private QueueConnectionFactory qconFactory;
private QueueConnection qcon;
private QueueSession qsession;
private QueueSender qsender;
private Queue queue;
private TextMessage msg;
```

Set up the JNDI initial context, as follows:

```
InitialContext ic = getInitialContext(args[0]);
       .
       .
       .
private static InitialContext getInitialContext(
 String url
) throws NamingException
{
 Hashtable env = new Hashtable();
 env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
 env.put(Context.PROVIDER_URL, url);
 return new InitialContext(env);
}
```

> **Note**
>
> When setting up the JNDI initial context for an EJB or servlet, use the following method:
>
> ```
> Context ctx = new InitialContext();
> ```

Create all the necessary objects for sending messages to a JMS queue. The `ctx` object is the JNDI initial context passed in by the `main()` method.

```
public void init(
 Context ctx,
 String queueName
) throws NamingException, JMSException
{
```

Step 1

Look up a connection factory in JNDI.

```
 qconFactory = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
```

Step 2

Create a connection using the connection factory.

```
 qcon = qconFactory.createQueueConnection();
```

Step 3

Create a session using the connection. The following code defines the session as non-transacted and specifies that messages will be acknowledged automatically. For more information about transacted sessions and acknowledge modes, see Session.

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

```
qsession = qcon.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
```

Step 4

Look up a destination (queue) in JNDI.

```
queue = (Queue) ctx.lookup(queueName);
```

Step 5

Create a reference to a message producer (queue sender) using the session and destination (queue).

```
qsender = qsession.createSender(queue);
```

Step 6

Create the message object.

```
msg = qsession.createTextMessage();
```

Step 7

Start the connection.

```
qcon.start();
}
```

The `init()` method for the `examples.jms.queue.QueueReceive` example is similar to the `QueueSend init()` method shown previously, with the one exception. Steps 5 and 6 would be replaced by the following code, respectively:

```
qreceiver = qsession.createReceiver(queue);
qreceiver.setMessageListener(this);
```

In the first line, instead of calling the `createSender()` method to create a reference to the queue sender, the application calls the `createReceiver()` method to create the queue receiver.

In the second line, the message consumer registers an asynchronous message listener.

When a message is delivered to the queue receiver, the provider calls `examples.jms.queue.QueueReceive.onMessage()`. The following code shows that callback method:

```
public void onMessage(Message msg)
{
 try {
       String msgText;
       if (msg instanceof TextMessage) {
          msgText = ((TextMessage)msg).getText();
       } else { // If it is not a TextMessage...
        msgText = msg.toString();
       }

       System.out.println("Message Received: "+ msgText );

       if ("quit".equalsIgnoreCase(msgText)) {
         synchronized(this) {

                quit = true;
              this.notifyAll(); // Notify main thread to quit
```


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

```
        }
      }
  } catch (JMSException jmse) {
   jmse.printStackTrace();
  }
}
```

The `onMessage()` method processes messages received through the queue receiver. For a `TextMessage`, it prints the text body. For another message type, it prints a diagnostic string representation. The null-safe comparison treats a text body equal to `quit`, ignoring case, as the request to stop the example.

> **Note**
>
> It is good practice to verify that the received message is the type expected by the handler method.

For more information about the JMS classes used in this example, see Understanding the JMS API or the `jakarta.jms` Javadoc, at http://www.oracle.com/technetwork/java/jms/index.html.

Example: Setting Up a Publish-Subscribe JMS Application Using the Classic API

The following example is an excerpt from the `examples.jms.topic.TopicSend` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/topic`, where `WL_HOME` is the WebLogic Server installation directory. The `init()` method shows you how to set up and start a topic session for a JMS application. The following shows the `init()` method, with comments describing each setup step.

Define the required variables, including the JNDI context, JMS connection factory, and topic static variables.

```
public final static String JNDI_FACTORY=
       "weblogic.jndi.WLInitialContextFactory";
public final static String JMS_FACTORY=
       "weblogic.examples.jms.TopicConnectionFactory";
public final static String
       TOPIC="weblogic.examples.jms.exampleTopic";

protected TopicConnectionFactory tconFactory;
protected TopicConnection tcon;
protected TopicSession tsession;
protected TopicPublisher tpublisher;
protected Topic topic;
protected TextMessage msg;
```

Set up the JNDI initial context, as follows:

```
InitialContext ic = getInitialContext(args[0]);
       .
       .
       .
private static InitialContext getInitialContext(
 String url
) throws NamingException
```


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

```
{
  Hashtable env = new Hashtable();
  env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
  env.put(Context.PROVIDER_URL, url);
  return new InitialContext(env);
}
```

> **Note**
>
> When setting up the JNDI initial context for a servlet, use the following method:
>
> `Context ctx = new InitialContext();`

Create all the necessary objects for sending messages to a JMS topic. The `ctx` object is the JNDI initial context passed in by the `main()` method.

```
public void init(
 Context ctx,
 String topicName
) throws NamingException, JMSException
{
```

Step 1

Look up a connection factory using JNDI.

```
  tconFactory =
   (TopicConnectionFactory) ctx.lookup(JMS_FACTORY);
```

Step 2

Create a connection using the connection factory.

```
  tcon = tconFactory.createTopicConnection();
```

Step 3

Create a session using the connection. The following defines the session as non-transacted and specifies that messages will be acknowledged automatically. For more information about setting session transaction and acknowledge modes, see Session.

```
  tsession = tcon.createTopicSession(false,
   Session.AUTO_ACKNOWLEDGE);
```

Step 4

Look up the destination (topic) using JNDI.

```
  topic = (Topic) ctx.lookup(topicName);
```

Step 5

Create a reference to a message producer (topic publisher) using the session and destination (topic).

```
  tpublisher = tsession.createPublisher(topic);
```

Step 6

Create the message object.

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

```
msg = tsession.createTextMessage();
```

Step 7

Start the connection.

```
tcon.start();
}
```

The `init()` method for the `examples.jms.topic.TopicReceive` example is similar to the `TopicSend init()` method shown previously with one exception. Steps 5 and 6 would be replaced by the following code, respectively:

```
tsubscriber = tsession.createSubscriber(topic);
tsubscriber.setMessageListener(this);
```

In the first line, instead of calling the `createPublisher()` method to create a reference to the topic publisher, the application calls the `createSubscriber()` method to create the topic subscriber.

In the second line, the message consumer registers an asynchronous message listener.

When a message is delivered to the topic subscriber, the provider calls `examples.jms.topic.TopicReceive.onMessage()`. This callback uses the same message-processing logic as `QueueReceive.onMessage()`, described in Example: Setting Up a Point-to-Point JMS Application Using the Classic API.

For more information about the JMS interfaces used in this example, see Understanding the JMS API or the `jakarta.jms` Javadoc, at `jakarta.jms package`.

Sending Messages

To send a message, you can use either the simplified API or the classic API.

You can start sending messages after you set up the JMS application as described in Setting Up a JMS Application.

Sending Messages Using the Simplified JMS API

In the simplified API, messages are sent by creating a `JMSProducer` object on behalf of `JMSContext`. For more information, see Create JMSProducer and JMSConsumer Objects.

To send a message to a specified destination, you can use the following `JMSProducer` method which is analogous to the `send` method of `MessageProducer` in the classic API:

```
JMSProducer send(Destination destination, Message message)
```

For example, to send an existing `Message` object:

```
context.createProducer().send(destination, message);
```

Alternatively, use the `String` overload to create and send a `TextMessage`:

```
context.createProducer().send(destination, "Hello");
```

The second call creates a `TextMessage` with the body `Hello` and sends it to the specified destination.

The following `JMSProducer.send` overloads create a message of the appropriate type and initialize its body from the supplied parameter:

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

```
JMSProducer send(Destination destination,byte[] body)

JMSProducer send(Destination destination, Map<String,Object> body)

JMSProducer send(Destination destination, Serializable body)

JMSProducer send(Destination destination, String body)
```

For more information about the `JMSProducer` interface and `send` methods, see the Javadoc at:

`JMSProducer`

Use the standard `JMSProducer` methods `setDeliveryMode`, `setPriority`, `setDeliveryDelay`, and `setTimeToLive` to configure message delivery. These methods do not require a WebLogic-specific cast. To use WebLogic extensions such as redelivery limits and send timeouts, cast a WebLogic JMS producer to `weblogic.jms.extensions.WLJMSProducer`. See `WLJMSProducer` in Java API Reference for Oracle WebLogic Server.

For example,

```
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT).send(destination,message);
```

A new `JMSProducer` uses `DeliveryMode.PERSISTENT`, priority `4`, and a time-to-live of `0` (no expiration). Use its setter methods to change these producer settings. Configured destination delivery overrides take precedence over the corresponding values supplied by the producer. See Setting JMSProducer and MessageProducer Attributes and Configure Resources for JMS System Modules in the Oracle WebLogic Remote Console Online Help.

Sending Messages Using the Classic JMS API

Once you have set up the JMS application as described in Using the Classic API to Set Up a JMS Application, you can send messages. To send a message, you must, in order, perform the steps described in the following sections:

For more information about the JMS interfaces for sending messages and the message types, see the `jakarta.jms.Message` Javadoc, at `Message`. For information about receiving messages, see Receiving Messages.

Create a Message Object

This step has already been completed as part of the client setup procedure, as described in Step 6a: Create the Message Object (Message Producers).

Define a Message

This step *may* have been completed when you set up an application, as described in Step 6a: Create the Message Object (Message Producers). Whether or not this step has already been completed depends on the method that was called to create the message object. For example, for TextMessage and ObjectMessage types, when you create a message object, you have the option of defining the message when you create the message object.

If the body already has the required value, continue to Send the Message to a Destination Using MessageProducer.

If no body value was specified, or if you want to change an existing value, use the appropriate setter method. For example, `TextMessage` provides the following method:

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

```
public void setText(
 String string
) throws JMSException
```

> **Note**
>
> A `TextMessage` or `ObjectMessage` can have a null body. A `Message` object passed to a send method must not be null.

Subsequently, you can clear the message body using the following method:

```
public void clearBody(
) throws JMSException
```

For more information about methods used to define messages, see the `jakarta.jms.Session` Javadoc, at `Session`.

Send the Message to a Destination Using MessageProducer

The `Destination` and `MessageProducer` objects were created when you set up the application, as described in Using the Classic API to Set Up a JMS Application.

> **Note**
>
> Each eligible topic subscription receives its own copy of a message. Consumers on the same shared subscription share that subscription's messages; each message is delivered to one consumer at a time. A received message's body and properties are read-only. Call `clearBody()` to clear the body and make it writable. Call `clearProperties()` separately to clear the properties and make them writable. Header fields are separate from the message body and properties.

For more information about the `MessageProducer` interface, see MessageProducer and MessageConsumer or the `jakarta.jms.MessageProducer` Javadoc, at `MessageProducer`.

Pass a `Message` object to `send`. For a producer created with a destination, use a send method without a destination argument. For an anonymous producer, created without a destination, supply a `Destination` object identifying the queue or topic.

The send overloads can specify delivery mode, priority, and time-to-live for an individual message. When those arguments are omitted, the producer's current values are used. Set delivery delay with `MessageProducer.setDeliveryDelay`; it is not a send-method argument.

WebLogic JMS initializes a classic `MessageProducer`'s delivery mode, priority, and time-to-live from the connection factory. Producer setter methods change those values for subsequent sends, while explicit send arguments apply to that send. Configured destination delivery overrides take precedence over the corresponding message values. See Setting JMSProducer and MessageProducer Attributes and Configure Resources for JMS System Modules in the Oracle WebLogic Remote Console Online Help.

Persistent messages use the persistent store associated with the hosting JMS server. Depending on the JMS server's targeting and configuration, it uses a configured custom store or the hosting WebLogic Server instance's default store. For custom-store configuration, see

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

Create a File Store and Create a JDBC Store in the Oracle WebLogic Remote Console Online Help.

> **Note**
>
> If `StoreEnabled` is false for the JMS server, persistent sends are rejected unless `AllowsPersistentDowngrade` is true. When that option is enabled, persistent messages are downgraded to `NON_PERSISTENT`. Omitting a custom store does not by itself disable persistence.

The domain-specific `QueueSender.send` and `TopicPublisher.publish` methods provide the corresponding queue and topic operations.

For additional information about the `QueueSender` interface methods, see the `jakarta.jms.QueueSender` Javadoc, at `QueueSender`.

For more information about the `TopicPublisher` interface methods, see the `jakarta.jms.TopicPublisher` Javadoc, at `TopicPublisher`.

Sending a Message Asynchronously

An asynchronous send can return control before the send has completed. The provider calls `CompletionListener.onCompletion(Message)` when the message has been successfully sent, with the same degree of confidence as a successful synchronous send. This callback does not mean that a consumer has received or acknowledged the message.

Implement both `onCompletion(Message)` and `onException(Message, Exception)`. A failure may be reported directly by the send call; if it cannot be thrown there, the provider reports it through `onException`. An exception does not by itself establish whether the message was sent. For more information, see the `CompletionListener` API reference.

Asynchronous sending with `CompletionListener` is not permitted in a Jakarta EE web or Enterprise Beans container. Use these examples in a supported client environment, such as a Java SE client.

After defining the `jakarta.jms.CompletionListener` object, send messages asynchronously using the `JMSProducer` or `MessageProducer` objects as described.

- If you are using `JMSProducer` objects to send messages, call the method `setAsync(CompletionListener listener)` with a non-null `CompletionListener` on the `JMSProducer` object before calling the `send` method as listed in the following example:

  ```
  // send a message asynchronously
  try (JMSContext context = connectionFactory.createContext()) {
   MyCompletionListener myCompletionListener = new MyCompletionListener();
  //call normal send method
   context.createProducer().setAsync(myCompletionListener).send(queue, "Hello world");
   ...
  }
  ```

  For more information, see Sending Messages Using the Simplified JMS API.

- If you are using a `MessageProducer` to send messages, use the following method to send messages asynchronously:

  ```
  messageProducer.send(message,completionListener);
  ```


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

For more information, see Sending Messages Using the Classic JMS API.

Setting JMSProducer and MessageProducer Attributes

WebLogic JMS initializes classic `MessageProducer` delivery settings from the connection factory. A new `JMSProducer` instead uses `DeliveryMode.PERSISTENT`, priority `4`, and a time-to-live of `0`. For WebLogic JMS, the delivery delay follows the connection factory's default time-to-deliver setting until the application sets a delivery delay on the producer. Producer setters change the values used for later sends. Configured destination delivery overrides take precedence over the corresponding message values. See Configure Resources for JMS System Modules in the Oracle WebLogic Remote Console Online Help.

The standard producer methods configure delivery mode, priority, time-to-live, and delivery delay. WebLogic producer extensions also configure redelivery limits and send timeouts. Redelivery delay is a separate session/context setting, available through `WLSession` or `WLJMSContext`; it is not a producer send timeout. Table 5-2 distinguishes the classic and simplified producer methods.

Each cell lists the setter followed by its getter. All methods in the classic API column declare `throws JMSException`. Simplified API setters return the producer for method chaining; their methods use unchecked JMS runtime exceptions. The first four rows use the standard interfaces. The last two rows require the WebLogic extension named in that column.

Table 5-2 Message Producer Set and Get Methods

Attribute	Classic API	Simplified API
Delivery mode	`MessageProducer`: `void setDeliveryMode(int deliveryMode)`; `int getDeliveryMode()`	`JMSProducer`: `JMSProducer setDeliveryMode(int deliveryMode)`; `int getDeliveryMode()`
Priority	`MessageProducer`: `void setPriority(int priority)`; `int getPriority()`	`JMSProducer`: `JMSProducer setPriority(int priority)`; `int getPriority()`
Time-to-live	`MessageProducer`: `void setTimeToLive(long timeToLive)`; `long getTimeToLive()`	`JMSProducer`: `JMSProducer setTimeToLive(long timeToLive)`; `long getTimeToLive()`
Delivery delay	`MessageProducer`: `void setDeliveryDelay(long deliveryDelay)`; `long getDeliveryDelay()`	`JMSProducer`: `JMSProducer setDeliveryDelay(long deliveryDelay)`; `long getDeliveryDelay()`
Redelivery limit	`WLMessageProducer`: `void setRedeliveryLimit(int redeliveryLimit)`; `int getRedeliveryLimit()`	`WLJMSProducer`: `WLJMSProducer setRedeliveryLimit(int redeliveryLimit)`; `int getRedeliveryLimit()`
Send timeout	`WLMessageProducer`: `void setSendTimeout(long sendTimeout)`; `long getSendTimeout()`	`WLJMSProducer`: `WLJMSProducer setSendTimeout(long sendTimeout)`; `long getSendTimeout()`

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

> **Note**
>
> JMS defines optional `MessageProducer` methods for disabling the message ID and timestamp information. However, these methods are ignored by WebLogic JMS.

For more information about the `MessageProducer` interface methods, see the `jakarta.jms.MessageProducer` Javadoc, at `MessageProducer`, or the `weblogic.jms.extensions.WLMessageProducer` Javadoc.

Example: Sending Messages Within a Point-to-Point Application

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/queue`, where `WL_HOME` is the WebLogic Server installation directory. The example shows the code required to create a `TextMessage`, set the text of the message, and send the message to a queue.

```
msg = qsession.createTextMessage();
       .
       .
       .
public void send(
 String message
) throws JMSException
{
 msg.setText(message);
 qsender.send(msg);
}
```

For more information about the `QueueSender` interface and methods, see the `jakarta.jms.QueueSender` Javadoc, at `QueueSender`.

Example: Sending Messages Within a Publish/Subscribe Application

The following example is excerpted from the `examples.jms.topic.TopicSend` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/topic`, where `WL_HOME` is the WebLogic Server installation directory. It shows the code required to create a `TextMessage`, set the text of the message, and send the message to a topic.

```
msg = tsession.createTextMessage();
       .
       .
       .
public void send(
 String message
) throws JMSException
{
 msg.setText(message);
 tpublisher.publish(msg);
}
```

For more information about the `TopicPublisher` interface and methods, see the `jakarta.jms.TopicPublisher` Javadoc, at `TopicPublisher`.

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

Receiving Messages

Learn how to receive messages using the `JMSConsumer` and `MessageConsumer` methods.

After you set up the JMS application as described in Setting Up a JMS Application, you can receive messages.

To receive a message, you must create the receiver object and specify whether you want to receive messages asynchronously or synchronously.

The order in which messages are received can be controlled by the following:

- Message delivery attributes (delivery mode and sorting criteria) defined during configuration or as part of the `send()` method, as described in Sending Messages.
- Destination sort order set using destination keys, as described in Configure Resources for JMS System Modules in the *Oracle WebLogic Remote Console Online Help*.

After the message is received, you can modify the header field values; however, the message properties and message body are read-only. You can modify the message body by executing the corresponding message type's `clearBody()` method to clear the existing contents and enable write permission.

For more information about the JMS interfaces for receiving messages and the message types, see the `jakarta.jms.Message` Javadoc, at `Message`. For information about sending messages, see Sending Messages.

Receive Messages Asynchronously Using the Simplified API

This procedure applies to Java SE applications and the Jakarta EE application client container. Web and EJB applications must use message-driven beans for asynchronous message receipt. See Jakarta EE Compliance.

To receive messages, you must first create a `JMSConsumer` object using one of the several `createConsumer` or `createDurableConsumer` methods on `JMSContext` object.

Create a `JMSConsumer` object and use the method `setMessageListener` to specify the object that implements the `MessageListener` interface. By default, creating the consumer starts message delivery automatically. If the application has called context.setAutoStart(false), call context.start() when it is ready to receive messages.

```
JMSConsumer consumer = context.createConsumer(queue);
consumer.setMessageListener(messageListener);
```

Receiving Messages Asynchronously using the Classic API

Receiving Messages Asynchronously using the Classic API is described within the context of setting up the application. For more information, see Step 6b: Optionally Register an Asynchronous Message Listener.

> **Note**
>
> You can control the maximum number of messages that may exist for an asynchronous consumer and that have not yet been passed to the message listener by setting the Messages Maximum attribute when configuring the connection factory.

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

Asynchronous Message Pipeline

For asynchronous consumption, WebLogic JMS can push messages to a consumer's client-side pipeline before its message listener is ready to process them. The message pipeline, or backlog, consists of messages accumulated for that consumer that have not yet been passed to its listener. The connection factory's Messages Maximum setting controls the pipeline size.

Configuring a Message Pipeline

You can control a client's maximum pipeline size by configuring the Messages Maximum per Session attribute on the client's connection factory, which is defined as the "maximum number of messages that can exist for an asynchronous consumer and that have not yet been passed to the message listener". The default setting is *10*. For more information about configuring a JMS connection factory, see Configure Resources for JMS System Modules in the *Oracle WebLogic Remote Console Online Help*.

Behavior of Pipelined Messages

After a message pipeline is configured, it will exhibit the following behavior:

- Statistics — JMS monitoring statistics reports backlogged messages in a message pipeline as pending (for queues and durable subscribers) until they are either committed or acknowledged.
- Performance — Increasing the Messages Maximum pipeline size may improve performance for high-throughput applications. Note that a larger pipeline will increase client memory usage as the pending pipelined messages accumulate on the client JVM before the asynchronous consumer's listener is called.
- Sorting — Messages in an asynchronous consumer's pipeline are not sorted according to the consumer destination's configured sort order; instead, they remain in the order in which they are pushed from the JMS server. For example, if a destination is configured to sort by priority, high priority messages will not jump ahead of low priority messages that have already been pushed into an asynchronous consumer's pipeline.

> **Note**
>
> The Messages Maximum per Session pipeline size setting on the connection factory is not related to the Messages Maximum quota settings on JMS servers and destinations.
>
> Messages in a pipeline can be aggregated into a single transport message. The combined encoded size must fit the effective maximum message size of the receiving network channel. The server's MaxMessageSize default is 10,000,000 bytes; a custom channel can override it with NetworkAccessPointMBean.MaxMessageSize. MaxT3MessageSize is a deprecated protocol-specific override. Account for aggregation and encoding overhead when choosing the pipeline size and message-size limits.

Receive Messages Synchronously Using the Simplified API

The `receive` methods on a `JMSConsumer` object are used for synchronous delivery of messages.

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

```
public String receiveMessage(
ConnectionFactory connectionFactory,Queue queue){
   String body=null;
   try (JMSContext context = connectionFactory.createContext();){
      JMSConsumer consumer = context.createConsumer(queue);
      body = consumer.receiveBody(String.class);
   } catch (JMSRuntimeException ex) {
      // handle exception
   }
   return body;
}
```

For additional information about the `JMSConsumer` interface methods, see the `jakarta.jms.JMSConsumer` Javadoc, at `JMSConsumer`.

Receiving Messages Synchronously Using the Classic API

To receive messages synchronously, use the following `MessageConsumer` methods:

```
public Message receive(
) throws JMSException

public Message receive(
 long timeout
) throws JMSException

public Message receiveNoWait(
) throws JMSException
```

`receive()` and `receive(0)` wait indefinitely for a message available to that consumer or until the consumer is closed. `receive(timeout)` waits up to the specified number of milliseconds and returns `null` if the timeout expires or the consumer is concurrently closed. `receiveNoWait()` returns an available message immediately, or `null` if none is available.

The `MessageConsumer` interface methods are inherited by the `QueueReceiver` and `TopicSubscriber` interfaces. For additional information about the `MessageConsumer` interface methods, see the `jakarta.jms.MessageConsumer` Javadoc, at `MessageConsumer`.

Example: Receiving Messages Synchronously Within a PTP Application

The following example is excerpted from the `examples.jms.queue.QueueReceive` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/queue`, where `WL_HOME` is the WebLogic Server installation directory. Rather than set a message listener, you would call `qreceiver.receive()` for each message. For example:

```
qreceiver = qsession.createReceiver(queue);
qreceiver.receive();
```

The first line creates the queue receiver on the queue. The second line executes a `receive()` method. The `receive()` method blocks and waits for a message.

Example: Receiving Messages Synchronously Within a Pub/Sub Application

The following example is excerpted from the `examples.jms.topic.TopicReceive` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/topic`, where

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

`WL_HOME` is the WebLogic Server installation directory. Rather than set a message listener, you would call `tsubscriber.receive()` for each message.

For example:

```
tsubscriber = tsession.createSubscriber(topic);
Message msg = tsubscriber.receive();
msg.acknowledge();
```

The first line creates the topic subscriber on the topic. The second line executes a `receive()` method. The `receive()` method blocks and waits for a message.

Use Prefetch Mode to Create a Synchronous Message Pipeline

Configure synchronous prefetch through SynchronousPrefetchMode on the connection factory's `ClientParamsBean`. The values are disabled (the default), enabled for queue receivers and topic subscribers, and topicSubscriberOnly for topic subscribers.

With prefetch enabled, WebLogic JMS can push messages ahead of synchronous receive calls so that messages are already available in the consumer's pipeline. MessagesMaximum controls the number of outstanding prefetched messages; it does not promise that every network batch has that size. Prefetch can reduce the number of synchronous server calls, but its performance benefit depends on the workload.

Prefetch is not supported for receives in a user (JTA/XA) transaction or with multiple consumers on one session. WebLogic JMS can disable prefetch when it detects these conditions before prefetch starts. After prefetch has started, a receive in a user transaction or an attempt to start prefetch on a second consumer can fail. Use one consumer per session and avoid user transactions when relying on synchronous prefetch.

For more information on the behavior of pipelined messages, see Asynchronous Message Pipeline. For more information on configuring a JMS connection factory, see Configure Resources for JMS System Modules in the *Oracle WebLogic Remote Console Online Help*.

Recovering Received Messages

> **Note**
>
> Session.recover() operates on a non-transacted session and is not limited to CLIENT_ACKNOWLEDGE. It requests redelivery of unacknowledged messages; it does not undo acknowledgments. A synchronously received AUTO_ACKNOWLEDGE message has already been acknowledged when receive returns.

Where this operation is permitted, call Session.recover() to restart delivery of the session's unacknowledged messages:

```
public void recover(
) throws JMSException
```

The `recover()` method performs the following steps:

- Stops message delivery for the session
- Tags all messages that have not been acknowledged (but may have been delivered) as redelivered

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

- Resumes sending messages starting from the first unacknowledged message for that session

> **Note**
>
> Messages in queues are not necessarily re delivered in the same order that they were originally delivered, nor to the same queue consumers. For information to guarantee the correct ordering of re delivered messages, see Ordered Redelivery of Messages.

Acknowledging Received Messages

Use the `acknowledge()` method to acknowledge a received message. This method depends on how the connection factory's Acknowledge Policy attribute is configured.

> **Note**
>
> This section applies only to non-transacted sessions for which the acknowledge mode is set to `CLIENT_ACKNOWLEDGE`.

To acknowledge a received message, use the following `Message` method:

```
public void acknowledge(
) throws JMSException
```

The `acknowledge()` method depends on how the connection factory's Acknowledge Policy attribute is configured, as follows:

- The default policy of "All" specifies that calling the acknowledge on a message acknowledges all unacknowledged messages received on the session.
- The "Previous" policy specifies that calling the acknowledge on a message acknowledges only unacknowledged messages up to, and including, the given message. Messages that are not acknowledged may be redelivered to the client.

This method is effective only when issued by a non-transacted session for which the acknowledge mode is set to `CLIENT_ACKNOWLEDGE`. Otherwise, the method is ignored.

Releasing Object Resources

Close application-managed JMS resources when they are no longer needed.

For the classic API, close Connection, Session, MessageProducer, MessageConsumer, ConnectionConsumer, and QueueBrowser objects when they are no longer needed. Their close methods have the following signature:

```
public void close(
) throws JMSException
```

For the simplified API, close an application-managed JMSContext, or create it in a try-with-resources statement. Closing the context closes its session and consumers, and closes its connection when no other open JMSContext uses it. A JMSConsumer can also be closed

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

separately. JMSProducer has no close method. Do not close an injected JMSContext; the container manages its lifecycle.

When closing an object:

- Outside message-listener and completion callbacks, closing a consumer, session, or connection waits for its pending receives and message listeners to finish. Closing a producer, session, or connection also waits for outstanding asynchronous sends and CompletionListener callbacks. Consult the API contract for restrictions on calling close from a callback.
- All associated sub objects are also closed. For example, when closing a session, all associated message producers and consumers are also closed. When closing a connection, all associated sessions are also closed.

For more information about the effects of the *`close()`* method for each object, see the appropriate `jakarta.jms` Javadoc, at `jakarta.jms package`. In addition, for more information about the connection or Session `close()` method, see Starting, Stopping, and Closing a Connection or Closing a Session, respectively.

The following example is an excerpt from the `examples.jms.queue.QueueReceive` example, provided with WebLogic Server. When the WebLogic Server examples are installed, its source is in `WL_HOME/samples/server/examples/src/examples/jms/classicapi/queue`, where `WL_HOME` is the WebLogic Server installation directory. This example shows the code required to close the message consumer, session, and connection objects.

```
public void close(
) throws JMSException
{
 qreceiver.close();
 qsession.close();
 qcon.close();
}
```

In the `QueueReceive` example, the `close()` method is called at the end of `main()` to close objects and free resources.

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

6

Managing Your Applications

Learn how to manage your JMS applications programmatically using the value-added WebLogic JMS features.

Managing Rolled Back, Recovered, Redelivered, or Expired Messages

Learn how to manage rolled back or recovered messages.

Setting a Redelivery Delay for Messages

You can delay the redelivery of messages when a temporary, external condition prevents an application from properly handling a message. This enables an application to temporarily inhibit the receipt of "poison" messages that it cannot currently handle. When a message is rolled back or recovered, the redelivery delay is the amount of time a message is put aside before an attempt is made to redeliver the message.

If JMS immediately redelivers the message, then the error condition may not be resolved and the application may still not be able to handle the message. However, if an application is configured for a redelivery delay, then when it rolls back or recovers a message, the message is set aside until the redelivery delay has passed, at which point the messages are made available for redelivery.

All messages consumed and subsequently rolled back or recovered by a session receive the redelivery delay for that session at the time of rollback or recovery. Messages consumed by multiple sessions as part of a single user transaction will receive different redelivery delays as a function of the session that consumed the individual messages. Messages that are left unacknowledged or uncommitted by a client, either intentionally or as a result of a failure, are not assigned a redelivery delay.

Setting a Redelivery Delay

A session inherits the redelivery delay from its connection factory when the session is created.

For more information, see Configure Resources for JMS System Modules in the *Oracle WebLogic Remote Console Online Help*.

The application that creates the session can then override the connection factory setting using WebLogic-specific extensions to the `jakarta.jms.Session` interface. The session attribute is dynamic and can be changed at any time. Changing the session redelivery delay affects all messages consumed and rolled back (or recovered) by that session after the change except when the message is in a session using non-durable topics.

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

> **Note**
>
> When a session is using non-durable topics, the `setRedeliveryDelay` method does not apply. This may result in unexpected behavior if you are using a non-durable topic consumer to drive a workflow.

The method for setting the redelivery delay on a session is provided through the `weblogic.jms.extensions.WLSession` interface, which is an extension to the `jakarta.jms.Session` interface. To define a redelivery delay for a session, use the following methods:

```
public void setRedeliveryDelay(
 long redeliveryDelay
) throws JMSException;

public long getRedeliveryDelay(
) throws JMSException;
```

For more information on the `WLSession` class, refer to the `weblogic.jms.extensions.WLSession` Javadoc.

Overriding the Redelivery Delay on a Destination

Regardless of what redelivery delay is set on the session, the destination where a message is being rolled back or recovered can override the setting. The redelivery delay override applied to the redelivery of a message is the one in effect at the time a message is rolled back or recovered.

Setting a Redelivery Limit for Messages

You can specify a limit on the number of times that WebLogic JMS will attempt to redeliver a message to an application. After WebLogic JMS fails to redeliver a message to a destination for a specific number of times, the message can be redirected to an error destination that is associated with the message destination. If the redelivery limit is configured, but no error destination is configured, then persistent or non-persistent messages are deleted when they reach their redelivery limit.

Alternatively, you can set the redelivery limit value dynamically using the message producer's set method, as described in Setting JMSProducer and MessageProducer Attributes.

Ordered Redelivery of Messages

> **Note**
>
> Oracle recommends that applications use the Ordered Redelivery upgrade to Message Unit-of-Order. See Using the Message Unit-of-Order.

All messages initially delivered to a consumer from a given producer are guaranteed to arrive at the consumer in the order in which they were produced. WebLogic JMS goes above and beyond this requirement by providing the "Ordered Redelivery of Messages" feature, which guarantees the correct ordering of *redelivered* messages *as well*.

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

In order to provide this guarantee, WebLogic JMS must impose certain constraints. They are:

- Single consumers — ordered redelivery is only guaranteed when there is a single consumer. If there are multiple consumers, then there are no guarantees about the order in which any individual consumer will receive messages.

> **Note**
>
> With respect to MDBs (message-driven beans), the number of consumers is a function of the number of MDB instances deployed for a given MDB. The initial and maximum values for the number of instances must be set to *1*. Otherwise no ordering guarantees can be made with respect to redelivered messages.

- Sort order : If a given destination is sorted, has JMS destination keys defined, and another message is produced such that the message would be placed at the top of the ordering, then no guarantee can be made between the redelivery of an existing message and the delivery of the incoming message.
- Message selection : If a consumer is using a selector, then ordering on redelivery is only guaranteed between the message being redelivered and other messages that match the criteria for that selector. There are no guarantees of order with respect to messages that do not match the selector.
- Redelivery delay : If a message has a redelivery delay period and is recovered or rolled back, then it is unavailable for the delay period. During that period, other messages can be delivered before the delayed message, even though these messages were sent after the delayed message.
- Messages pending recovery : Ordered redelivery does not apply to redelivered messages that end up in a pending recovery state due to a server failure or a system reboot.

Required Message Pipeline Setting for the Messaging Bridge and MDBs

For asynchronous consumers or JMS applications using the WebLogic Messaging Bridge or MDBs, the size of the message pipeline must be set to *1*. The pipeline size is set using the Messages Maximum attribute on the JMS connection factory used by the receiving application. Any value higher than *1* means there may be additional in-flight messages that will appear ahead of a redelivered message. MDB applications must define an application-specific JMS connection factory and set the Messages Maximum attribute value to *1* on that connection factory, and then reference the connection factory in the EJB descriptor for their MDB application.

For more information about programming EJBs, see Message-Driven EJBs in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Performance Limitations

JMS applications that implement the Ordered Redelivery feature will incur performance degradation for asynchronous consumers using JTA transactions (specifically, MDBs and the WebLogic Messaging Bridge). This is caused by a mandatory reduction in the number of in-flight messages to exactly 1, so messages are not aggregated when they are sent to the client.

Handling Expired Messages

WebLogic JMS has an *active* message Expiration Policy feature that allows you to control how the system searches for expired messages and how it handles them when they are

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

encountered. This feature ensures that expired messages are cleaned up immediately, either by simply discarding expired messages, discarding expired messages and logging their removal, or redirecting expired messages to an error destination configured on the local JMS server.

Setting Message Delivery Times

You can schedule message deliveries to an application for specific times in the future.

Message deliveries can be deferred for short periods of time (such as seconds or minutes) or for long stretches of time (for example, hours later for batch processing). Until that delivery time, the message is essentially invisible until it is delivered, enabling you to schedule work at a particular time in the future.

Messages are not sent on a recurring basis; they are sent only once. In order to send messages on a recurring basis, a received scheduled message must be sent back to its original destination. Typically, the receive, the send, and any associated work should be under the same transaction to ensure exactly-once semantics.

Setting a Delivery Time on Producers

Support for setting and getting a time-to-deliver on an individual producer is provided through the `weblogic.jms.extensions.WLMessageProducer` interface, which is an extension to the `jakarta.jms.MessageProducer` interface. To define a time-to-deliver on an individual producer, use the following methods:

```
public void setTimeToDeliver(
 long timeToDeliver
) throws JMSException;

public long getTimeToDeliver(
) throws JMSException;
```

For more information about the `WLMessageProducer` class, see the `weblogic.jms.extensions.WLMessageProducer` Javadoc.

Setting a Delivery Time on Messages

`DeliveryTime` is a JMS message header field that defines the earliest absolute time at which the message can be delivered. That is, the message is held by the messaging system and is not given to any consumers until that time.

As a JMS header field, `DeliveryTime` can be used to sort messages in a destination or to select messages. For the purposes of data type conversion, the delivery time is stored as a long integer.

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

> **Note**
>
> Setting a delivery time value on a message has no effect on this field, because JMS will always override the value with the producer's value when the message is sent or published. The message delivery time methods described here are similar to other JMS message fields that are set through the producer, including the delivery mode, priority, time-to-deliver, time-to-live, redelivery delay, and redelivery limit fields. Specifically, the setting of these fields is reserved for JMS providers, including WebLogic JMS.

The support for setting and getting the delivery time on a message is provided through the `weblogic.jms.extensions.WLMessage` interface, which is an extension to the `jakarta.jms.Message` interface. To define a delivery time on a message, use the following methods:

```
public void setJMSDeliveryTime(
 long deliveryTime
) throws JMSException;

public long getJMSDeliveryTime(
) throws JMSException;
```

For more information about the `WLMessage` class, see `weblogic.jms.extensions.WLMessage` Javadoc.

Overriding a Delivery Time

When a producer is created it inherits its `TimeToDeliver` attribute, expressed in milliseconds, from the connection factory used to create the connection that the producer is a part of. Regardless of the time-to-deliver set on the producer, the destination to which a message is being sent or published can override the setting. An administrator can set the `TimeToDeliverOverride` attribute on a destination in either a relative or scheduled string format.

Interaction with the Time-to-Live Value

If the specified time-to-live value (`JMSExpiration`) is less than or equal to the specified time-to-deliver value, then the message delivery succeeds. However, the message is then silently expired.

Setting a Scheduled Time-to-Deliver Override

The scheduled `TimeToDeliverOverride` attribute can also be specified using the `weblogic.jms.extensions.Schedule` class, which provides methods that take a schedule and return the next scheduled time for delivering messages.

Table 6-1 Message Delivery Schedule

Example	Description
`0 0 0,30 * * * *`	Exact next nearest half-hour
`* * 0,30 4-5 * * *`	Anytime in the first minute of the half hours between 4 A.M. and 5 A.M. hours

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

Table 6-1 (Cont.) Message Delivery Schedule

Example	Description
`* * * 9-16 * * *`	Between 9 A.M. and 5 P.M. (9:00.00 A.M. to 4:59.59 P.M.)
`* * * * 8-14 * 2`	The second Tuesday of the month
`* * * 13-16 * * 0`	Between 1 P.M. and 5 P.M. on Sunday
`* * * * * 31 *`	The last day of the month
`* * * * 15 4 1`	The next time April 15th occurs on a Sunday
`0 0 0 1 * * 2-6;0 0 0 2 * * 1,7`	1 A.M. on weekdays; 2 A.M. on weekends

A cron-like string is used to define the schedule. The format is defined by the following BNF syntax:

```
schedule := millisecond second minute hour dayOfMonth month
      dayOfWeek
```

The BNF syntax for specifying the `second` field is as follows:

```
second   := * | secondList
secondList := secondItem [, secondList]
secondItem := secondValue | secondRange
SecondRange := secondValue - secondValue
```

Similar BNF statements for milliseconds, minute, hour, day of month, month, and day of week can be derived from the second syntax. The values for each field are defined as non-negative integers in the following ranges:

```
milliSecondValue := 0-999
milliSecondValue := 0-999
secondValue   := 0-59
minuteValue   := 0-59
hourValue    := 0-23
dayOfMonthValue := 1-31
monthValue    := 1-12
dayOfWeekValue  := 1-7
```

> **Note**
>
> These values equate to the same ranges that the`java.util.Calendar` class uses, except for `monthValue`. The `java.util.Calendar` range for `monthValue` is 0-11, rather than 1-12.

Using this syntax, each field can be represented as a range of values indicating all times between the two times. For example, `2-6` in the `dayOfWeek` field indicates Monday through Friday, inclusive. Each field can also be specified as a comma-separated list. For instance, a minute field of `0,15,30,45` means every quarter hour on the quarter hour. Last, each field can be defined as both a set of individual values and ranges of values. For example, an hour field of `9-17,0` indicates between the hours of 9 A.M. and 5 P.M., and on the hour of midnight.

Additional semantics are as follows:

- If multiple schedules are supplied (using a semi-colon (;) as the separator), then the next scheduled time for the set is determined using the schedule that returns the soonest value.

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

One use for this is for specifying schedules that change based on the day of the week (see the example below).

- A value of 1 (one) for `dayOfWeek` equates to Sunday.
- A value of * means every time for that field. For instance, a * in the Month field means every month. A * in the Hour field means every hour.
- A value of `l` or `last` (not case sensitive) indicates the greatest possible value for a field.
- If a day of the month is specified that exceeds the normal maximum for a month, then the normal maximum for that month will be specified. For example, if it is February during a leap year and 31 was specified, then the scheduler will schedule as if 29 was specified instead. This means that setting the month field to 31 always indicates the last day of the month.
- If milliseconds are specified, then they are rounded down to the nearest 50th of a second. The values are 0, 19, 39, 59, ..., 979, and 999. Thus, 0-40 gets rounded to 0-39 and 50-999 gets rounded to 39-999.

> **Note**
>
> When a calendar is not supplied as a method parameter to one of the static methods in this class, the calendar used is a `java.util.GregorianCalendar` with a default `java.util.TimeZone` and a default `java.util.Locale`.

JMS Schedule Interface

The `weblogic.jms.extensions.Schedule` class has methods that will return the next scheduled time that matches the recurring time expression. This expression uses the same syntax as `TimeToDeliverOverride`. The time returned in milliseconds can be relative or absolute.

For more information about the `Schedule` class, see `weblogic.jms.extensions.Schedule` Javadoc.

You can define the next scheduled time after the *given* time using the following method:

```
public static Calendar nextScheduledTime(String schedule, Calendar calendar) throws
ParseException;
```

You can define the next scheduled time after the current time using the following method:

```
public static Calendar nextScheduledTime(String schedule) throws ParseException;
```

You can define the next scheduled time after the given time in absolute milliseconds using the following method:

```
public static long nextScheduledTimeInMillis(String schedule, long timeInMillis) throws
ParseException;
```

You can define the next scheduled time after the given time in relative milliseconds using the following method:

```
public static long nextScheduledTimeInMillisRelative(String schedule, long timeInMillis)
throws ParseException;
```

You can define the next scheduled time after the current time in relative milliseconds using the following method:

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

```
public static long nextScheduledTimeInMillisRelative(String schedule) throws
ParseException;
```

Managing Connections

Learn how to manage JMS connections.

Defining a Connection Exception Listener

An exception listener asynchronously notifies an application whenever a problem occurs with a connection. This mechanism is particularly useful for a connection waiting to consume messages that might not be notified otherwise.

> **Note**
>
> The purpose of an exception listener is not to monitor all exceptions thrown by a connection, but to deliver those exceptions that would not be otherwise delivered.

You can define an exception listener for a connection using the following `Connection` method:

```
public void setExceptionListener(
 ExceptionListener listener
) throws JMSException
```

You must specify an `ExceptionListener` object for the connection.

The JMS Provider notifies an exception listener, if one has been defined, when it encounters a problem with a connection using the following `ExceptionListener` method:

```
public void onException(
 JMSException exception
)
```

The JMS provider specifies the exception that describes the problem when calling the method.

You can access the exception listener for a connection using the following Connection method:

```
public ExceptionListener getExceptionListener(
) throws JMSException
```

Accessing Connection Metadata

You can access the metadata associated with a specific connection using the following `Connection` method:

```
public ConnectionMetaData getMetaData() throws JMSException
```

This method returns a `ConnectionMetaData` object that enables you to access JMS metadata. The following table lists the various type of JMS metadata and the get methods that you can use to access them.

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

Table 6-2 JMS Metadata

JMS Metadata	Get Method
Version	`public String getJMSVersion() throws JMSException`
Major version	`public int getJMSMajorVersion() throws JMSException`
Minor version	`public int getJMSMinorVersion() throws JMSException`
Provider name	`public String getJMSProviderName() throws JMSException`
Provider version	`public String getProviderVersion() throws JMSException`
Provider major version	`public int getProviderMajorVersion() throws JMSException`
Provider minor version	`public int getProviderMinorVersion() throws JMSException`
JMSX property names	`public Enumeration getJMSXPropertyNames() throws JMSException`

For more information about the `ConnectionMetaData` class, see the `jakarta.jms.ConnectionMetaData` Javadoc at `ConnectionMetaData`.

Starting, Stopping, and Closing a Connection

To control the flow of messages, you can start and stop a connection temporarily using the `start()` and `stop()` methods, respectively, as follows.

The `start()` and `stop()` method details are as follows:

```
public void start() throws JMSException

public void stop() throws JMSException
```

A newly created connection is stopped—no messages are received until the connection is started. Typically, other JMS objects are set up to handle messages before the connection is started, as described in Setting Up a JMS Application. Messages may be produced on a stopped connection, but cannot be delivered to a stopped connection.

Once started, you can stop a connection using the `stop()` method. This method performs the following steps:

- Pauses the delivery of all messages. No applications waiting to receive messages will return until the connection is restarted or the time-to-live value associated with the message is reached.
- Waits until all message listeners that are currently processing messages have completed.

Typically, a JMS Provider allocates a significant amount of resources when it creates a connection. When a connection is no longer being used, you should close it to free up resources. A connection can be closed using the following method:

```
public void close(
) throws JMSException
```

This method performs the following steps to execute an orderly shutdown:

- Terminates the receipt of all pending messages. Applications may return a message or null if a message was not available at the time of the close.
- Waits until all message listeners that are currently processing messages have completed.

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

- Rolls back in-process transactions on its transacted sessions (unless such transactions are part of an external JTA user transaction). For more information about JTA user transactions, see Using JTA User Transactions.
- Does not force an acknowledge of client-acknowledged sessions. By not forcing an acknowledge, no messages are lost for queues and durable subscriptions that require reliable processing.

When you close a connection, all associated objects are also closed. You can continue to use the message objects created or received via the connection, except the received message's `acknowledge()` method. Closing a closed connection has no effect.

> **Note**
>
> Attempting to acknowledge a received message from a closed connection's session throws an `IllegalStateException`.

Managing Sessions

Learn how to manage JMS sessions.

Defining a Session Exception Listener

An exception listener asynchronously notifies a client in the event a problem occurs with a session. This is particularly useful for a session waiting to consume messages that might not be notified otherwise.

> **Note**
>
> The purpose of an exception listener is not to monitor all exceptions thrown by a session, only to deliver those exceptions that would otherwise be undelivered.

You can define an exception listener for a session using the following `WLSession` method:

```
public void setExceptionListener(ExceptionListener listener) throws JMSException
```

You must specify an `ExceptionListener` object for the session.

The JMS Provider notifies an exception listener, if one has been defined, when it encounters a problem with a session using the following `ExceptionListener` method:

```
public void onException(JMSException exception)
```

The JMS Provider specifies the exception encountered that describes the problem when calling the method.

You can access the exception listener for a session using the following `WLSession` method:

```
public ExceptionListener getExceptionListener() throws JMSException
```


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

> **Note**
>
> Because there can only be one thread per session, an exception listener and message listener (used for asynchronous message delivery) cannot execute simultaneously. Consequently, if a message listener is executing at the time a problem occurs, execution of the exception listener is blocked until the message listener completes its execution. For more information about message listeners, see Receiving Messages Asynchronously using the Classic API.

Closing a Session

As with connections, a JMS provider allocates a significant amount of resources when it creates a session. When a session is no longer being used, it is recommended that it be closed to free up resources. A session can be closed using the following `Session` method:

```
public void close(
) throws JMSException
```

> **Note**
>
> The `close()` method is the only Session method that can be invoked from a thread that is separate from the session thread.

This method does the following to execute an orderly shutdown:

- Terminates the receipt of all pending messages. Applications can return a message or null if a message was not available at the time connection was closed.
- Waits until all message listeners that are currently processing messages have completed.
- Rolls back in-process transactions (unless these transactions are part of external JTA user transaction). For more information about JTA user transactions, see Using JTA User Transactions.
- Does not force an acknowledgement of client acknowledged sessions, ensuring that no messages are lost for queues and durable subscriptions that require reliable processing.

When you close a session, all associated producers and consumers are also closed.

> **Note**
>
> If you want to issue the `close()` method within an `onMessage()` method call, then the system administrator must select the Allow Close In OnMessage check box when configuring the connection factory.

Managing Destinations

Learn how to create and delete JMS destinations.

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

Dynamically Creating Destinations

See the following topics for information about creating destinations dynamically:

- Using JMS Module Helper to Manage Applications briefs you about how to use the `weblogic.jms.extensions.JMSModuleHelper`. For more information about Using JMS Module Helper, see Using JMS Module Helper to Manage Applications
- Using Temporary Destinations briefs you about how applications are enabled to create destinations as per requirement. For more information about Using Temporary Destinations, seeUsing Temporary Destinations

The associated procedures for creating dynamic destinations are described in the following sections.

Dynamically Deleting Destinations

You can dynamically delete JMS destinations (queue or topic) using any of the following methods:

- JMSModuleHelper class (see Using JMS Module Helper to Manage Applications)
- User-defined JMX application

The JMS server removes the deleted destination in real time, therefore, it is not necessary to redeploy the JMS server for the deletion to take effect.

Required Conditions for Deleting Destinations

In order to successfully delete a destination, the following conditions must be met:

- The destination must not be a member of a distributed destination. For more information, see Using Distributed Destinations.
- The destination must not be the error destination for some other destination.

If either of these conditions cannot be met, then the deletion will not be allowed.

What Happens when a Destination Is Deleted

When a destination is deleted, the following behaviors and semantics apply:

- Physical deletion of existing messages : All durable subscribers for the deleted destination are permanently deleted. All messages, persistent and non-persistent, stored in the deleted destination are permanently removed from the messaging system.
- No longer able to create producers, consumers, and browsers : After a destination is deleted, applications will no longer be able to create producers, consumers, or browsers for the deleted destination. Any attempt to do so will result in the application receiving an `InvalidDestinationException` — as if the destination does not exist.
- Closing of consumers : All existing consumers for the deleted destination are closed. The closing of a consumer generates a `ConsumerClosedException`, which is delivered to the `ExceptionListener`, if any, of the parent session, and which will read "Destination was deleted".

  When a consumer is closed, if it has an outstanding `receive()` operation, then that operation is cancelled and the caller receives a `null` value indicating that no message is

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

available. Attempts by an application to do anything but `close()` a closed consumer will result in an `IllegalStateException`.

- Closing of browsers: All browsers for the deleted destination are closed. Attempts by an application to do anything but `close()` a closed browser will result in an `IllegalStateException`. Closing of a browser implicitly closes all enumerations associated with the browser.
- Closing of enumerations : All enumerations for the deleted destination are closed. The behavior after an enumeration is closed depends on the last call before the enumeration was closed. If a call to `hasMoreElements()` returns a value of *true*, and no subsequent call to `nextElement()` has been made, then the enumeration guarantees that the next element can be enumerated. This produces the specifics. When the last call before the close was to `hasMoreElements()`, and the value returned was *true*, then the following behaviors apply:
  - The first call to the `nextElement()` will return a message.
  - Subsequent calls to the `nextElement()` will throw a `NoSuchElementException`.
  - Calls to the`hasMoreElements()` made before the first call to the `nextElement()` will return true.
  - Calls to the `hasMoreElements()` made after the first call to the `nextElement()` will return false.

  If a given enumeration was never called, or the last call before the close was to `nextElement()`, or the last call before the close was to the `hasMoreElements()` and the value returned was false, then the following behaviors apply:
  - Calls to the`hasMoreElements()` will return false.
  - Calls to the `nextElement()` will throw a `NoSuchElementException`.
- Blocking send operations cancelled — all blocking send operations posted against the deleted destination are cancelled. Send operations waiting for quota will receive a `ResourceAllocationException`.
- Uncommitted transactions unaffected : The deletion of a destination does not affect existing uncommitted transactions. Any uncommitted work associated with a deleted destination is allowed to complete as part of the transaction. However, because the destination is deleted, the net result of all operations (rollback, commit, and so on) is the deletion of the associated messages.

Message Timestamps for Troubleshooting Deleted Destinations

If a destination with persistent messages is deleted and then immediately re-created while the JMS server is not running, then the JMS server will compare the version number of the destination (using the `CreationTime` field in the configuration `config.xml` file) to the version number of the destination in the persistent messages. In this case, the left over persistent messages for the older destination will have an older version number than the version number in the `config.xml` file for the re-created destination, and when the JMS server is rebooted, the left over persistent messages are discarded.

However, if a persistent message somehow has a version number that is *newer* than the version number in the `config.xml` for the re-created destination, then either the system clock was rolled back when the destination was deleted and re-created (while the JMS server was not running), or a different `config.xml` is being used. In this situation, the JMS server will fail to boot. To save the persistent message, you can set the version number (the `CreationTime` field) in the `config.xml` to match the version number in the persistent message. Otherwise, you can change the version number in the `config.xml` so that it is newer than the version number in the persistent message; this way, the JMS server can delete the message when it is rebooted.

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

Deleted Destination Statistics

Statistics for the deleted destination and the hosting JMS server are updated as the messages are physically deleted. However, the deletion of some messages can be delayed pending the outcome of another operation. This includes messages sent and received in a transaction, as well as unacknowledged non-transactional messages received by a client.

Using Temporary Destinations

Temporary destinations are enabled by default through the JMS server's `Hosting Temporary Template` attribute. They enable an application to create a destination, as required, without the system administration overhead associated with configuring and creating a server-defined destination.

JMS applications can use the `JMSReplyTo` header field to return a response to a request. The sender application may optionally set the `JMSReplyTo` header field of its messages to its temporary destination name to advertise the temporary destination that it is using to other applications.

Temporary destinations exist only for the duration of the current connection, unless they are removed using the `delete()` method, described in Deleting a Temporary Destination.

Because messages are never available if the server is restarted, all `PERSISTENT` messages are silently made `NON_PERSISTENT`. As a result, temporary destinations are not suitable for business logic that must survive a restart.

The following sections describe how to create a temporary queue (Point-to-Point) or temporary topic (Publish/Subscibe).

Creating a Temporary Queue

You can create a temporary queue using the following `QueueSession` method:

```
public TemporaryQueue createTemporaryQueue(
) throws JMSException
```

For example, to create a reference to a `TemporaryQueue` that will exist only for the duration of the current connection, use the following method call:

```
QueueSender = Session.createTemporaryQueue();
```

Creating a Temporary Topic

You can create a temporary topic using the following `TopicSession` method:

```
public TemporaryTopic createTemporaryTopic(
) throws JMSException
```

For example, to create a reference to a temporary topic that will exist only for the duration of the current connection, use the following method call:

```
TopicPublisher = Session.createTemporaryTopic();
```


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

Deleting a Temporary Destination

When you finish using a temporary destination, you can delete it (to release associated resources) using the following `TemporaryQueue` or `TemporaryTopic` method:

```
public void delete(
) throws JMSException
```

Setting Up Durable Subscriptions

WebLogic JMS supports durable and non durable subscriptions. Learn how to set up durable subscriptions for your application.

For durable subscriptions, WebLogic JMS stores a message in a persistent file or database until the message is delivered to the subscribers or has expired, even if those subscribers are not *active* at the time that the message is delivered. A subscriber is considered active if the Java object that represents it exists. Durable subscriptions are supported for Publish/Subscribe messaging only.

> **Note**
>
> Durable subscriptions cannot be created for distributed topics. However, you can still create a durable subscription on distributed topic member and the other topic members will forward the messages to the member that has the durable subscription. See Using Distributed Destinations.

For non durable subscriptions, WebLogic JMS delivers messages only to applications with an active session. Messages sent to a topic while an application is not listening are never delivered to that application. In other words, non durable subscriptions last only as long as their subscriber objects. By default, subscribers are non durable.

The following sections describe:

Defining the Persistent Store

You must configure a persistent file or database store and assign it to your JMS server so WebLogic JMS can store a message until it is delivered to the subscribers or has expired.

- Create a JMS file store or JMS JDBC backing store using the Stores node.
- Target the configured store to your JMS server by selecting it from the Store field's drop-down list on the General tab of the configuration page under JMS Server.

> **Note**
>
> No two JMS servers can use the same backing store.

Setting the Client ID Policy

The Client ID Policy specifies whether more than one JMS connection can use the same client ID in a cluster. Valid values for this policy are:

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

- `RESTRICTED`: The default. Only one connection that uses this policy can exist in a cluster at any given time for a particular client ID (If a connection already exists with a given Client ID, attempts to create new connections using this policy with the same client ID fail with an exception).
- `UNRESTRICTED`: Connections created using this policy can specify any Client ID, even when other restricted or unrestricted connections already use the same client ID. When a durable subscription is created using an Unrestricted client ID, it can only be cleaned up using `weblogic.jms.extensions.WLJMSContext.unsubscribe(Topic topic, String name)` or using `weblogic.jms.extensions.WLSession.unsubscribe(Topic topic, String name)`. See Managing Durable Subscriptions.

Oracle recommends setting the client ID policy to `Unrestricted` for new applications (unless your application architecture requires exclusive client IDs), especially if sharing a subscription (durable or non-durable). Subscriptions created with different client ID policies are always treated as independent subscriptions. See `ClientIdPolicy` in the *MBean Reference for Oracle WebLogic Server*.

For more information about advanced concepts and functionality of Uniform Distributed Topics (UDTs) necessary to design high availability applications, see Shared Subscriptions and Client ID Policy.

Defining the Client ID

To support durable subscriptions, a client identifier (client ID) must be defined for the connection.

> **Note**
>
> The JMS client ID is not necessarily equivalent to the WebLogic Server username, that is, a name used to authenticate a user in the WebLogic security realm. You can set the JMS client ID to the WebLogic Server username, if it is appropriate for your JMS application.

The client ID can be supplied in two ways:

- The first method is to configure the connection factory with the client ID. For WebLogic JMS, this means adding a separate connection factory definition during configuration for each client ID. Applications then look up their own topic connection factories in JNDI and use them to create connections that contain their own client IDs.
- Alternatively, the preferred method is for an application that can set its client ID in the connection after the connection is created by calling the following connection method:

  ```
  public void setClientID(
   String clientID
  ) throws JMSException
  ```

  If you use this alternative approach, then you can use the default connection factory (if it is acceptable for your application) and avoid the need to modify the configuration information. However, applications with durable subscriptions must ensure that they call the`setClientID()` method *immediately after* creating their topic connection.

  If a client ID is already defined for the connection, then an `IllegalStateException` is thrown. If the specified client ID is already defined for another connection, then an `InvalidClientIDException` is thrown.

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

> **Note**
>
> When specifying the client ID using the `setClientID()` method, there is a risk that a duplicate client ID may be specified without throwing an exception. For example, if the client IDs for two separate connections are set simultaneously to the same value, then a race condition may occur and the same value may be assigned to both connections. You can avoid this risk of duplication by specifying the client ID during configuration.

To display a client ID and test whether or not a client ID has been defined already, use the following connection method:

```
public String getClientID(
) throws JMSException
```

> **Note**
>
> Support for durable subscriptions is a feature unique to the Publish/Subscribe messaging model, so client IDs are used only with topic connections; queue connections also contain client IDs, but JMS does not use them.
>
> Durable subscriptions should not be created for a temporary topic, because a temporary topic is designed to exist only for the duration of the current connection.

Creating a Sharable Subscription Policy

The Subscription Sharing policy specifies whether subscribers can share subscriptions with other subscribers on the same connections on this connection. Valid values for this policy are:

- `Exclusive`: The default. All subscribers created using this connection factory cannot share subscriptions with any other subscribers. Use this policy to retain the functionality of WebLogic Server 10.3.4.0 and earlier.
- `Sharable`: Subscribers created using this connection factory can share their subscriptions with other subscribers, regardless of whether those subscribers are created using the same connection factory or a different connection factory. Consumers can share non durable subscriptions only if they have the same client ID and client ID policy; consumers can share a durable subscription only if they have the same client ID, client ID policy, and subscription name.

WebLogic JMS applications can override the Subscription Sharing policy specified on the connection factory configuration by casting a `jakarta.jms.JMSContext` instance to `weblogic.jms.extensions.WLJMSContext` or a `jakarta.jms.Connection` instance to `weblogic.jms.extensions.WLConnection` and calling `setSubscriptionSharingPolicy(String subscriptionSharingPolicy)`.

Most applications with a Sharable Subscription Sharing policy will also use an Unrestricted client ID policy in order to ensure that multiple connections with the same client ID can exist.

Two durable subscriptions with the same client ID and subscription name are treated as two different independent subscriptions if they have a different Client ID Policy. Similarly, two Sharable non durable subscriptions with the same client ID are treated as two different independent subscriptions if they have a different client ID policy.

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

For more information on how to use the Subscription Sharing policy, see Shared Subscriptions and Client ID Policy.

Creating Subscribers for a Durable Subscription

This section describes how to create subscribers for a durable subscription and contains the following topics:

Using JMS 2.0 API

To create subscribers for an unshared durable subscription use one of the following methods:

```
public MessageConsumer createDurableConsumer(
 Topic topic,
 String name
) throws JMSException

or

public MessageConsumer createDurableConsumer(
 Topic topic,
 String name,
 String selector,
 boolean noLocal
) throws JMSException
```

Using JMS 1.1 API

You can create subscribers for a durable subscription using the following `TopicSession` methods:

```
public TopicSubscriber createDurableSubscriber(
 Topic topic,
 String name
) throws JMSException

or

public TopicSubscriber createDurableSubscriber(
 Topic topic,
 String name,
 String messageSelector,
 boolean noLocal
) throws JMSException
```

You must specify the name of the topic for which you are creating a subscriber and the name of the durable subscription.

> **Note**
>
> Valid durable subscription names cannot include the following characters: comma , equals, colon , asterisk , percent , or question mark.

You may also specify a message selector for filtering messages and a `noLocal` flag (described later in this section). Message selectors are described in more detail in Filtering Messages. If you do not specify a `selector` or `messageSelector`,then by default all messages are searched.

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

An application can use a JMS connection to both publish and subscribe to the same topic. Because topic messages are delivered to all subscribers, an application can receive messages it has published itself. To prevent this, a JMS application can set a `noLocal` flag to `true`. The default for the `noLocal` value is `false`. Durable subscriptions are stored within the file or database.

Best Practice: Always Close Failed JMS ClientIDs

As a best practice, JMS clients should always call the `close()` method instead of allowing the application to rely on the JVM's garbage collection to clean failed JMS connections. This is particularly important for durable subscription ClientIDs because the JMS Automatic Reconnect feature keeps a reference to failed JMS connections. Therefore, always use `connection.close()` method to clean up your connections. Also, consider using a `finally` block to ensure that your connection resources are cleaned up. Otherwise, WebLogic Server allocates system resources to keep the connection available.

The following code example demonstrates using the`close()` method and the`finally` block in a JMS client to clean up failed connection resources:

```
JMSConnection con = null;
try {
  con = cf.createConnection();
  con.setClientID("Fred");
  // Do some I/O stuff;
}
finally {
  if (con != null) con.close();
}
```

For more information about the JMS Automatic Reconnect feature, see Automatic JMS Client Failover.

Deleting Durable Subscriptions

To delete a durable subscription, you use the following `TopicSession` method:

```
public void unsubscribe(
 String name
) throws JMSException
```

You must specify the name of the durable subscription to be deleted.

You cannot delete a durable subscription if any of the following are true:

- A `TopicSubscriber` is still active on the session.
- A message received by the durable subscription is part of a transaction or has not yet been acknowledged in the session.

  For information about managing durable subscriptions, see Managing Durable Subscriptions.

Modifying Durable Subscriptions

To modify a durable subscription, perform the following steps:

1. Delete the durable subscription, as described in Deleting Durable Subscriptions.

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

If it is not explicitly performed, the deletion will be executed implicitly when the durable subscription is recreated in the next step.

2. Use the methods described in Creating Subscribers for a Durable Subscription to re-create a durable subscription of the same name, but specifying a different topic name, message selector, or `noLocal` value.

   The durable subscription is re-created based on the new values.

> **Note**
>
> When re-creating a durable subscription, be careful to avoid creating a durable subscription with a duplicate name. For example, if you attempt to delete a durable subscription from a JMS server that is unavailable, the delete call fails. If you subsequently create a durable subscription with the same name on a different JMS server, you may experience unexpected results when the first JMS server becomes available. Because the original durable subscription has not been deleted, when the first JMS server again becomes available, there will be two durable subscriptions with duplicate names.

Managing Durable Subscriptions

You can monitor and manage durable topic subscribers through public runtime APIs. This functionality also enables you to view and browse *all* messages, and to manipulate *most* messages on durable subscribers. This includes message browsing (for sorting), message manipulation (such as move and delete), and message import and export. For more information, see and Managing JMS Messages in *Administering JMS Resources for Oracle WebLogic Server*.

Setting and Browsing Message Header and Property Fields

WebLogic JMS provides a set of standard header fields that you can define to identify and route messages. In addition, property fields enable you to include application-specific header fields within a message, extending the standard set. You can use the message header and property fields to convey information between communicating processes.

The primary reason for including data in a property field rather than in the message body is to support message filtering through message selectors. Except for XML message extensions, data in the message body cannot be accessed through message selectors. For example, suppose you use a property field to assign high priority to a message. You can then design a message consumer that contains a message selector that accesses this property field and selects only messages of expedited priority. See Filtering Messages.

Setting Message Header Fields

JMS messages contain a standard set of header fields that are always transmitted with the message. They are available to message consumers that receive messages, and some fields can be set by the message producers that send messages. After a message is received, its header field values can be modified.

When modifying (overriding) header field values, you must take into consideration instances when message fields are overwritten by the JMS subsystem. For instance, setting the priority on a producer affects the priority of the message, but a value supplied to the `send()` method overrides the setting on the producer. Similarly, values set on a destination override values set

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

by the producer or values supplied to the `send()` method. The only way to verify the value of header fields is to query the message after a `send()` method.

For a description of the standard messages header fields, see Message Header Fields.

Table 6-3 lists the message class set and get methods for each of the supported data types.

> **Note**
>
> In some cases, the `send()` method overrides the header field value set using the `set()` method, as indicated in the following table.

Table 6-3 JMS Header Field Methods

Header Field	Set Method	Get Method
`JMSCorrelationID`	`public void setJMSCorrelationID(String correlationID) throws JMSException`	`public String getJMSCorrelationID() throws JMSException` `public byte[] getJMSCorrelationIDAsBytes() throws JMSException`
`JMSDestination`[1]	`public void setJMSDestination(Destination destination) throws JMSException`	`public Destination getJMSDestination() throws JMSException`
`JMSDeliveryMode1`	`public void setJMSDeliveryMode(int deliveryMode) throws JMSException`	`public int getJMSDeliveryMode() throws JMSException`
`JMSDeliveryTime1`	`public void setJMSDeliveryTime(long deliveryTime) throws JMSException`	`public long getJMSDeliveryTime() throws JMSException`
`JMSExpiration1`	`public void setJMSExpiration(long expiration) throws JMSException`	`public long getJMSExpiration() throws JMSException`

`JMSMessageID1`	`public void setJMSMessageID(String id) throws JMSException`
	Note: In addition to the set method, the `weblogic.jms.extensions.JMSRuntimeHelper` class provides the following methods to convert between pre-WebLogic JMS 6.0 and 6.1 `JMSMessageID` formats:
	`public void oldJMSMessageIDToNew(String id, long timeStamp) throws JMSException`
	`public void newJMSMessageIDToOld(String id, long timeStamp) throws JMSException`
		`public String getJMSMessageID() throws JMSException`

`JMSPriority1`	`public void setJMSPriority(int priority) throws JMSException`	`public int getJMSPriority() throws JMSException`
`JMSRedelivered1`	`public void setJMSRedelivered(boolean redelivered) throws JMSException`	`public boolean getJMSRedelivered() throws JMSException`
`JMSRedeliveryLimit1`	`public void setJMSRedeliveryLimit(int redelivered) throws JMSException`	`public int getJMSRedeliveryLimit() throws JMSException`

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

Table 6-3 (Cont.) JMS Header Field Methods

Header Field	Set Method	Get Method
`JMSReplyTo`	`public void setJMSReplyTo(Destination replyTo) throws JMSException`	`public Destination getJMSReplyTo() throws JMSException`
`JMSTimeStamp`[1]	`public void setJMSTimeStamp(long timestamp) throws JMSException`	`public long getJMSTimeStamp() throws JMSException`
`JMSType`	`public void setJMSType(String type) throws JMSException`	`public String getJMSType() throws JMSException`

[1] The corresponding `set()` method has no impact on the message header field when the `send()` method is executed. If set, this header field value will be overridden during the `send()` operation.

The `examples.jms.sender.SenderServlet` example, provided with WebLogic Server in the *EXAMPLES_HOME*`\wl_server\examples\src\examples\jms\sender` directory, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured, shows how to set header fields in messages that you send and how to display message header fields after they are sent.

For example, the following code, which appears after the `send()` method, displays the message ID that was assigned to the message by WebLogic JMS:

```
System.out.println("Sent message " + msg.getJMSMessageID() + " to " +
msg.getJMSDestination());
```

Setting Message Property Fields

To set a property field, call the appropriate set method and specify the property name and value. To read a property field, call the appropriate `get method` and specify the property name.

The sending application can set properties in the message, and the receiving application can subsequently view them. The receiving application cannot change the properties without first clearing them using the following `clearProperties()` method:

```
public void clearProperties(
) throws JMSException
```

This method does not clear the message header fields or body.

> **Note**
>
> The `JMSX` property name prefix is reserved for JMS. The connection metadata contains a list of `JMSX` properties, which can be accessed as an enumerated list using the `getJMSXPropertyNames()` method. For more information, see Accessing Connection Metadata.
>
> The `JMS_` property name prefix is reserved for provider-specific properties; it is not intended for use with standard JMS messaging.

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

The property field can be set to any of the following types: `boolean, byte, double, float, int, long, short, or string`. The following table lists the `Message class set` and `get methods` for each of the supported data types.

Table 6-4 Message Property Set and Get Methods for Data Types

Data Type	Set Method	Get Method
boolean	`public void setBooleanProperty(String name, boolean value) throws JMSException`	`public boolean getBooleanProperty(String name) throws JMSException`
byte	`public void setByteProperty(String name, byte value) throws JMSException`	`public byte getByteProperty(String name) throws JMSException`
double	`public void setDoubleProperty(String name, double value) throws JMSException`	`public double getDoubleProperty(String name) throws JMSException`
float	`public void setFloatProperty(String name, float value) throws JMSException`	`public float getFloatProperty(String name) throws JMSException`
int	`public void setIntProperty(String name, int value) throws JMSException`	`public int getIntProperty(String name) throws JMSException`
long	`public void setLongProperty(String name, long value) throws JMSException`	`public long getLongProperty(String name) throws JMSException`
short	`public void setShortProperty(String name, short value) throws JMSException`	`public short getShortProperty(String name) throws JMSException`
String	`public void setStringProperty(String name, String value) throws JMSException`	`public String getStringProperty(String name) throws JMSException`

In addition to the `set` and `get methods` described in the previous table, you can use the `setObjectProperty()` and `getObjectProperty()` methods to use the objectified primitive values of the property type. When the objectified value is used, the property type can be determined at execution time rather than during the compilation. The valid object types are Boolean, Byte, Double, Float, Integer, Long, Short, and String. For a property set with `setObjectProperty`, use the conversion row for the value's runtime type.

You can access all property field names using the following Message method:

```
public Enumeration getPropertyNames(
) throws JMSException
```

This method returns all property field names as an enumeration. You can then retrieve the value of each property field by passing the property field name to the appropriate `get method`, as described in the Table 6-4, based on the property field data type.

Table 6-5 contains a conversion chart for message properties. It enables you to identify the type that can be read based on the type that has been written. For each property type listed in the left-most column in which a message has been written, a **YES** in one of the remaining columns indicates that the message can be read as the type listed at the top of that column.

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

Table 6-5 Message Property Conversion Chart

Property Written As. . .	boolean	byte	double	float	int	long	short	String
boolean	**YES**	No	No	No	No	No	No	**YES**
byte	No	**YES**	No	No	**YES**	**YES**	**YES**	**YES**
double	No	No	**YES**	No	No	No	No	**YES**
float	No	No	**YES**	**YES**	No	No	No	**YES**
int	No	No	No	No	**YES**	**YES**	No	**YES**
long	No	No	No	No	No	**YES**	No	**YES**
short	No	No	No	No	**YES**	**YES**	**YES**	**YES**
String	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**

You can test whether or not a property value was set using the following `Message` method:

```
public boolean propertyExists(
 String name
) throws JMSException
```

You specify a property name and the method returns a Boolean value indicating whether or not the property exists.

For example, the following code sets two `String` properties and an `int` property:

```
msg.setStringProperty("User", user);
msg.setStringProperty("Category", category);
msg.setIntProperty("Rating", rating);
```

For more information about message property fields, see Message Property Fields , or the `jakarta.jms.Message` Javadoc at `Message`.

Browsing Header and Property Fields

> **Note**
>
> Only queue message header and property fields can be browsed. You cannot browse topic message header and property fields.

You can browse the header and property fields of messages on a queue using the following `QueueSession` methods:

```
public QueueBrowser createBrowser(
 Queue queue
) throws JMSException

public QueueBrowser createBrowser(
 Queue queue,
 String messageSelector
) throws JMSException
```


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

You must specify the queue that you want to browse. You can also specify a message selector to filter messages that you are browsing. Message selectors are described in more detail in Filtering Messages.

After you define a queue, you can access the queue name and message selector associated with a queue browser using the following `QueueBrowser` methods:

```
public Queue getQueue(
) throws JMSException

public String getMessageSelector(
) throws JMSException
```

In addition, you can access an enumeration for browsing the messages using the following `QueueBrowser` method:

```
public Enumeration getEnumeration(
) throws JMSException
```

The `examples.jms.queue.QueueBrowser` example, provided with WebLogic Server in the *EXAMPLES_HOME*`\wl_server\examples\src\examples\jms\queue` directory, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured, shows how to access the header fields of received messages.

For example, the following code is an excerpt from the `QueueBrowser` example and creates the `QueueBrowser` object:

```
qbrowser = qsession.createBrowser(queue);
```

The following is an excerpt from the `displayQueue()` method defined in the `QueueBrowser` example. In this example, the `QueueBrowser` object is used to obtain an enumeration that is subsequently used to scan the queue's messages.

```
  public void displayQueue(
  ) throws JMSException
 {
  Enumeration e = qbrowser.getEnumeration();
  Message m = null;

  if (! e.hasMoreElements()) {
   System.out.println("There are no messages on this queue.");
  } else {

   System.out.println("Queued JMS Messages: ");
   while (e.hasMoreElements()) {
    m = (Message) e.nextElement();
    System.out.println("Message ID " + m.getJMSMessageID() +
             " delivered " + new Date(m.getJMSTimestamp())
             " to " + m.getJMSDestination());
   }
  }
```

When a queue browser is no longer being used, you should close it to free up resources. For more information, see Releasing Object Resources.

For more information about the `QueueBrowser` class, see the `jakarta.jms.QueueBrowser` Javadoc at `QueueBrowser`.

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

Filtering Messages

In many cases, an application does not need to be notified of every message that is delivered to it. Use message selectors to filter unwanted messages, and subsequently improve performance by minimizing their effect on network traffic.

Message selectors operate as follows:

- The sending application sets message header or property fields to describe or classify a message in a standardized way.
- The receiving applications specify a simple query string to filter the messages that they want to receive.

Because message selectors cannot reference the contents (body) of a message, some information may be duplicated in the message property fields (except in the case of XML messages).

You specify a selector when creating a queue receiver or topic subscriber, as an argument to the `QueueSession.createReceiver()` or `TopicSession.createSubscriber()` methods, respectively. For information about creating queue receivers and topic subscribers, see Step 5: Create Message Producers and Message Consumers.

WebLogic JMS assigns a state or current processing condition to messages during processing. You can use these states as selectors. For information on valid message states, see weblogic.jms.extensions.JMSMessageInfo in *Java API Reference for Oracle WebLogic Server*.

The following sections describe how to define a message selector using SQL statements and XML selector methods, and how to update message selectors. For more information about setting header and property fields, see Setting and Browsing Message Header and Property Fields and Setting Message Property Fields, respectively.

Defining Message Selectors Using SQL Statements

A message selector is a Boolean expression. It consists of a String with a syntax similar to the `where` clause of an `SQL select` statement.

The following excerpts provide examples of selector expressions.

```
salary > 64000 and dept in ('eng','qa')

(product like 'WebLogic%' or product like '%T3')
       and version > 3.0

hireyear between 1990 and 1992
       or fireyear is not null

fireyear - hireyear > 4
```

The following example shows how to set a selector when creating a queue receiver that filters out messages with a priority lower than 6.

```
String selector = "JMSPriority >= 6";
qsession.createReceiver(queue, selector);
```

The following example shows how to set the same selector when creating a topic subscriber.

```
String selector = "JMSPriority >= 6";
qsession.createSubscriber(topic, selector);
```


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

For more information about the message selector syntax, see the `jakarta.jms.Message` Javadoc at `Message`.

Defining XML Message Selectors Using XML Selector Method

For XML message types, in addition to using the SQL selector expressions described in the previous section to define message selectors, you can use the following method:

```
String JMS_BEA_SELECT(String type, String expression)
```

The `JMS_BEA_SELECT` is a built-in function in WebLogic JMS SQL syntax. You specify the syntax type, which must be set to `xpath` (XML path language) and an XPath expression. The XML path language is defined in the XML Path Language (XPath) document, which is available at the XML Path Language web site at: `http://www.w3.org/TR/xpath`.

> **Note**
>
> Pay careful attention to your XML message syntax, since malformed XML messages (for example, a missing end tag) will not match any XML selector.

The method returns a null value under the following circumstances:

- The message does not parse.
- The message parses, but the element is not present.
- If a message parses and the element is present, but the message contains no value (for example, `<order></order>`).

For example, consider the following XML code example:

```
<order>
       <item>
              <id>007</id>
              <name>Hand-held Power Drill</name>
              <description>Compact, assorted colors.</description>
              <price>$34.99</price>
       </item>
       <item>
              <id>123</id>
              <name>Mitre Saw</name>
              <description>Three blades sizes.</description>
              <price>$69.99</price>
       </item>
       <item>
              <id>66</id>
              <name>Socket Wrench Set</name>
              <description>Set of 10.</description>
              <price>$19.99</price>
       </item>
</order>
```

The following example shows how to retrieve the name of the second item in the previous example. This method call returns the string, `Mitre Saw`.

```
String sel = "JMS_BEA_SELECT('xpath', '/order/item[2]/name/text()') = 'Mitre Saw'";
```


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

Pay careful attention to the use of double and single quotation mark and spaces. Note the use of single quotation mark around `xpath`, the XML tab, and the string value.

The following example shows how to retrieve the ID of the third item in the previous example. This method call returns the string, `66`.

```
String sel = "JMS_BEA_SELECT('xpath', '/order/item[3]/id/text()') = '66'";
```

Displaying Message Selectors

You can use the following `MessageConsumer` method to display a message selector:

```
public String getMessageSelector(
) throws JMSException
```

This method returns either the currently defined message selector or null if a message selector is not defined.

Indexing Topic Subscriber Message Selectors to Optimize Performance

For a certain class of applications, WebLogic JMS can significantly optimize topic subscriber message selectors by indexing them. These applications typically have a large number of subscribers, each with a unique identifier (like a user name), and they need to be able to quickly send a message to a single subscriber or to a list of subscribers. A typical example is an instant messaging application where each subscriber corresponds to a different user, and each message contains a list of one or more target users.

To activate optimized subscriber message selectors, subscribers must use the following syntax for their selectors:

```
"identifier IS NOT NULL"
```

`identifier` is an arbitrary string that is not a predefined JMS message property (e.g., neither `JMSCorrelationID` nor `JMSType`). Multiple subscribers can share the same identifier.

WebLogic JMS uses this message selector syntax as a hint to build internal subscriber indexes. Message selectors that do not follow the syntax, or that include additional `OR` and `AND` clauses, are still honored, but do not activate the optimization.

After subscribers register using this message selector syntax, a message published to the topic can target specific subscribers by including one or more identifiers in the message's user properties, as shown in the following example:

```
// Set up a named subscriber, where "wilma" is the name of
// the subscriber and subscriberSession is a JMS TopicSession.
// Note that the selector syntax used activates the optimization.

TopicSubscriber topicSubscriber =
  subscriberSession.createSubscriber(
    (Topic)context.lookup("IMTopic"),
    "Wilma IS NOT NULL",
    /* noLocal= */ true);

// Send a message to subscribers "Fred" and "Wilma",
// where publisherSession is a JMS TopicSession. Subscribers
// with message selector expressions "Wilma IS NOT NULL"
// or "Fred IS NOT NULL" will receive this message.

TopicPublisher topicPublisher =
 publisherSession.createPublisher(
```


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

```
  (Topic)context.lookup("IMTopic");

TextMessage msg =
 publisherSession.createTextMessage("Hi there!");
msg.setBooleanProperty("Fred", true);
msg.setBooleanProperty("Wilma", true);

topicPublisher.publish(msg);
```

> **Note**
>
> The optimized message selector and message syntax is based on the standard JMS API; therefore, applications that use this syntax will also work on versions of WebLogic JMS that do not have optimized message selectors, and on non-WebLogic JMS products. However, these versions will not perform as well as versions that include this enhancement.
>
> The message selector optimization will have no effect on applications that use the `MULTICAST_NO_ACKNOWLEDGE` acknowledge mode. These applications have no need for the enhancement anyway, because the message selection occurs on the client side rather than on the server side.

Sending XML Messages

The WebLogic Server JMS API provides native support for the Document Object Model (DOM) to send XML messages.

> **Note**
>
> This release does not support streaming. Only text and DOM representations of XML documents are supported.

The following sections provide information on WebLogic JMS API extensions that provide enhanced support for XML messages.

WebLogic XML APIs

You can use the following WebLogic XML APIs for transformation of XML between `String` and `DOM` representations:

- `XMLMessage`: Use to send messages with XML content.
- `WLSession.createXMLMessage` : Use to create an XML message.

It is possible for the payload of `XMLMessage` to be set using one XML representation and retrieved using a different representation. For example, it is valid for the XMLMessage body to be set using a `String` representation and be retrieved using a DOM representation.

Using a String Representation

Use the following steps to publish an XML message using a `string` type:

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

1. Serialize the XML to a `StringWriter`.
2. Call `StringWriter.toString()` and pass the result to `message.setText`.
3. Publish the message.

Using a DOM Representation

Sending XML messages using a DOM representation provides a significant performance improvement over sending messages as a `String`. Use the following steps to publish an XML message using a DOM representation:

1. If necessary, generate a DOM document from your XML source.
2. Pass the DOM document into the `XMLMessage.setDocument`.
3. Publish the message.

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

7

Using JMS Module Helper to Manage Applications

Learn how to create and manage JMS servers, Store-and-Forward agents, and JMS system resources by using `JMSModuleHelper`.

See `weblogic.jms.extensions.JMSModuleHelper`.

Configuring JMS System Resources Using JMSModuleHelper

You can manage a system module, including the JMS resources it contains by providing the domain MBean or by providing the initial context to the administration server in the API signatures defined by the `JMSModuleHelper` class.

The `JMSModuleHelper` class provides the following API signatures to manage a system module and JMS resources, such as queues and topics:

- Create a resource
- Create and modify resource
- Delete a resource
- Find and modify a resource
- Find using a template

See Configuring Basic JMS System Resources in the *Administering JMS Resources for Oracle WebLogic Server*.

Configuring JMS Servers and Store-and-Forward Agents

You can manage JMS servers and Store-and-Forward agents by providing the domain MBean or by providing the initial context to the administration server in the API signature defined by the `JMSModuleHelper` class.

The `JMSModuleHelper` class provides the following method APIs to manage JMS servers and Store-and-Forward agents:

- Create JMS servers and Store-and-Forward Agents
- Delete JMS servers and Store-and-Forward Agents
- Deploy JMS servers and Store-and-Forward Agents
- Undeploy JMS servers and Store-and-Forward Agents

Related Topics

- Configuring Basic JMS System Resources in the *Administering JMS Resources for Oracle WebLogic Server*.
- Understanding the Store-and-Forward Service in the *Administering JMS Resources for Oracle WebLogic Server*.

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

JMSModuleHelper Sample Code

Learn how to create and delete a JMS system resource module by following instructions in the sample code.

Creating a JMS System Resource

The module contains a connection factory and a topic.

Example 7-1 shows how to create JMS system resources.

Example 7-1 Create JMS System Resources

```
.
.
.
private static void createJMSUsingJMSModuleHelper(Context ctx){
System.out.println(
     "\n\n.... Configure JMS Resource for C API Topic Example ....\n\n");

     try {

     MBeanHome mbeanHome =
        (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
        DomainMBean domainMBean = mbeanHome.getActiveDomain();
        String domainMBeanName = domainMBean.getName();
        ServerMBean[] servers = domainMBean.getServers();

      String jmsServerName = "examplesJMSServer";

//
// create a JMSSystemResource "CapiTopic-jms"
//
     String resourceName = "CapiTopic-jms";
     JMSModuleHelper.createJMSSystemResource(
        ctx,
         resourceName,
         servers[0].getName());
         JMSSystemResourceMBean jmsSR =
              JMSModuleHelper.findJMSSystemResource(
                    ctx,
                    resourceName);
          JMSBean jmsBean = jmsSR.getJMSResource();
     System.out.println("Created JMSSystemResource " + resourceName);

//
// create a JMSConnectionFactory "CConFac"
//
     String factoryName = "CConFac";
     String jndiName = "CConFac";
     JMSModuleHelper.createConnectionFactory(
        ctx,
        resourceName,
         factoryName,
         jndiName,
          servers[0].getName());
     JMSConnectionFactoryBean factory =
jmsBean.lookupConnectionFactory(factoryName);
     System.out.println("Created Factory " + factory.getName());
```


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

```
//
// create a topic "CTopic"
//
     String topicName = "CTopic";
     String topicjndiName = "CTopic";
     JMSModuleHelper.createTopic(
        ctx,
        resourceName,
        jmsServerName,
        topicName,
        topicjndiName);

     TopicBean topic = jmsBean.lookupTopic(topicName);
     System.out.println("Created Topic " + topic.getName());
     } catch (Exception e) {
         System.out.println("Example configuration failed :" + e.getMessage());
        e.printStackTrace();
     }
}
.
.
.
```

Deleting a JMS System Resource

The following code removes JMS system resources.

Example 7-2 shows how to delete the JMS system resources.

Example 7-2 Delete JMS System Resources

```
.
.
.
private static void deleteJMSUsingJMSModuleHelper(Context ctx ) {

     System.out.println("\n\n.... Remove JMS System Resource for C API Topic
Example ....\n\n");


     try {

        MBeanHome mbeanHome =
           (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
           DomainMBean domainMBean = mbeanHome.getActiveDomain();
           String domainMBeanName = domainMBean.getName();
            ServerMBean[] servers = domainMBean.getServers();

     String jmsServerName = "examplesJMSServer";

//
// delete JMSSystemResource "CapiTopic-jms"
//
     String resourceName = "CapiTopic-jms";
     JMSModuleHelper.deleteJMSSystemResource(
        ctx,
        resourceName
         );
     } catch (Exception e) {
        System.out.println("Example configuration failed :" + e.getMessage());
        e.printStackTrace();
     }
```


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

```
}
.
.
.
```

Security Considerations for Anonymous Users

If your application environment depends on using anonymous users, you can create a security role for Anonymous and then apply a policy to the `weblogic.management.mbeanservers` JNDI resource that allows access by users in that role.

See Security for WebLogic Server MBeans in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

Since WebLogic Server 10.3.6, the JMSModuleHelper does not support anonymous lookup (using `-Dweblogic.management.anonymousAdminLookupEnabled=true`) to comply with the existing WebLogic security model.

Best Practices When Using JMSModuleHelper

Understand the best practices to follow when using the `JMSModuleHelper` class to configure JMS servers and resources.

- Check whether MBean objects (such as servers, JMS servers, or modules) are null before trying to use them.
- A create or delete method call can fail without throwing an exception. In addition, a thrown exception does not necessarily indicate that the method call failed.
- The time required to create the destination on the JMS server and propagate the information to the JNDI namespace can be significant. The propagation delay increases if the environment contains multiple servers. It is recommended that you test for the existence of the queue or topic, respectively, using the session `createQueue()` or `createTopic()` method, rather than perform a JNDI lookup. By doing so, you can avoid some of the propagation-specific delay.

  For example, the following method, findQueue(), attempts to access a dynamically created queue. Both retryCount and retryInterval must be nonnegative. It makes one initial attempt and up to retryCount retries, pausing for retryInterval milliseconds between attempts. It does not pause after the final failure. If the wait is interrupted, it restores the thread's interrupt status and throws a JMSException.

```
private static Queue findQueue (
 QueueSession queueSession,
 String jmsServerName,
 String queueName,
 int retryCount,
 long retryInterval
) throws JMSException
{
  String wlsQueueName = jmsServerName + "/" + queueName;
  String command = "QueueSession.createQueue(" +
       wlsQueueName + ")";
  long startTimeMillis = System.currentTimeMillis();
  if (retryCount < 0 || retryInterval < 0) {
    throw new JMSException("retryCount and retryInterval must be nonnegative");
}
for (int i = retryCount; i >= 0; i--) {
    try {
        System.out.println("Trying " + command);
```


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

```
            Queue queue = queueSession.createQueue(wlsQueueName);
            System.out.println(command + " succeeded after " +
                (retryCount - i + 1) + " tries in " +
                (System.currentTimeMillis() - startTimeMillis) + " millis.");
            return queue;
        } catch (JMSException je) {
            if (i == 0) throw je;
        }
        if (i > 0) {
            try {
                System.out.println(command + "> failed, pausing " +
                    retryInterval + " millis.");
                Thread.sleep(retryInterval);
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
                JMSException interrupted = new JMSException(
                    "Interrupted while waiting to retry queue lookup.");
                interrupted.setLinkedException(ie);
                throw interrupted;
            }
        }
    }
    throw new JMSException("out of retries");
    }
```

You can then call the `findQueue()` method after the `JMSModuleHelper` class method call to retrieve the dynamically created queue after it becomes available. For example:

```
JMSModuleHelper.createPermanentQueueAsync(ctx, domain, jmsServerName,
 queueName, jndiName);
Queue queue = findQueue(qsess, jmsServerName, queueName,
 retry_count, retry_interval);
```


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

8

Using Multicasting with WebLogic JMS

Learn how WebLogic JMS Multicasting enables the delivery of messages to a select group of hosts that subsequently forward the messages to subscribers in a cluster.

Benefits of Using Multicasting

Understand the benefits of using multicasting.

- Near real-time delivery of messages to a host group
- High scalability due to the reduction in the amount of resources required by the JMS server to deliver messages to topic subscribers in a cluster

Limitations of Using Multicasting

Understand the limitations of multicasting and the scenarios when multicasting should not be used.

The limitations of multicasting include:

- Multicast messages are not guaranteed to be delivered to all members of the host group. For messages requiring reliable delivery and recovery, you should not use multicasting.
- For interoperability with different versions of WebLogic Server, clients cannot have an earlier release of WebLogic Server installed than the host has. They must all have at least the same version or later.

For an example of when multicasting might be useful, consider a stock ticker. When accessing stock quotes, timely delivery is more important than reliability. When accessing the stock information in real-time, if all or a portion of the contents is not delivered, the client can request the information to be resent. Clients would not want to have the information recovered, in this case, as by the time it is redelivered, it would be out-of-date.

Using WebLogic Server Unicast

WebLogic Server provides an alternative to using multicast to handle cluster messaging and communications. Unicast configuration is much easier because it does not require the cross network configuration that multicast requires. Additionally, it reduces potential network errors that can occur from multicast address conflicts.

JMS topics configured for multicasting can access WebLogic clusters configured for unicast because a JMS topic publishes messages on its own multicast address that is independent of the cluster address. However, the following considerations apply:

- The router hardware configurations that allow unicast clusters may not allow JMS multicast subscribers to work.
- JMS multicast subscribers need to be in a network hardware configuration that allows multicast accessibility.

See Communications In a Cluster in *Administering Clusters for Oracle WebLogic Server*.

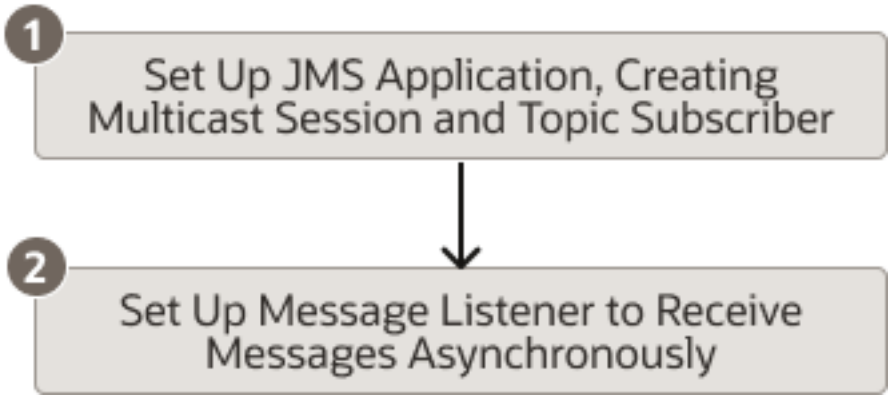
Copyright © 2007, 2026, Oracle and/or its affiliates.

Configuring Multicasting for WebLogic Server

Learn how to configure multicasting for WebLogic server.

Figure 8-1 shows the steps required to set up multicasting.

Figure 8-1 Setting Up Multicasting



> **Note**
>
> Multicasting is only supported for the Publish/Subscribe messaging model, and only for non durable subscribers.
>
> Monitoring statistics are not provided for multicast sessions or consumers.

Prerequisites for Multicasting

Before setting up multicasting, the connection factory and destination must be configured to support multicasting, as follows:

- For each connection factory, the system administrator configures the maximum number of outstanding messages that can exist on a multicast session and whether the most recent or oldest messages are discarded in the event the maximum is reached. If the message maximum is reached, a `DataOverrunException` is thrown, and messages are automatically discarded. These attributes are also dynamically configurable, as described in Dynamically Configuring Multicasting Configuration Attributes.
- For each destination, the Multicast Address (IP), Port, and TTL (Time-To-Live) attributes are specified. To better understand the TTL attribute setting, see Example: Multicast Time-to-Live.

> **Note**
>
> It is strongly recommended that you seek the advice of your network administrator when configuring the multicast IP address, port, and time-to-live attributes to ensure that the appropriate values are set.

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

Step 1: Set Up the JMS Application, Multicast Session and Topic Subscriber

Set up the JMS application as described in Setting Up a JMS Application. However, when creating sessions, as described in Step 3: Create a Session Using the Connection, specify that the session would like to receive multicast messages by setting the *acknowledgeMode* value to `MULTICAST_NO_ACKNOWLEDGE`.

> **Note**
>
> Multicasting is only supported for the Publish/Subscribe messaging model for non-durable subscribers. An attempt to create a durable subscriber on a multicast session will cause a `JMSException` to be thrown.

For example, the following method shows how to create a multicast session for the Publish/Subscribe messaging model.

This fragment assumes that `weblogic.jms.extensions.WLSession` is imported.

```
tsession = tcon.createTopicSession(false,
    WLSession.MULTICAST_NO_ACKNOWLEDGE);
```

> **Note**
>
> On the client side, each multicasting session requires one dedicated thread to retrieve messages off the socket. Therefore, you should increase the JMS client-side thread pool size to adjust for this.

In addition, create a topic subscriber, as described in Create TopicPublishers and TopicSubscribers.

For example, the following code illustrates how to create a topic subscriber:

```
tsubscriber = tsession.createSubscriber(myTopic);
```

> **Note**
>
> The `createSubscriber()` method fails if the specified destination is not configured to support multicasting.

Step 2: Set Up the Message Listener

Multicast topic subscribers can only receive messages asynchronously. If you attempt to receive synchronous messages on a multicast session, then a `JMSException` is thrown.

Set up the message listener for the topic subscriber, as described in Receiving Messages Asynchronously using the Classic API.

For example, the following code shows how to establish a message listener:

```
tsubscriber.setMessageListener(this);
```


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

When receiving messages, WebLogic JMS tracks the order in which messages are sent by the destinations. If a multicast subscriber's message listener receives the messages out of sequence, resulting in one or more messages being skipped, then a `SequenceGapException` will be delivered to the `ExceptionListener` for the session(s) present. If a skipped message is subsequently delivered, then it will be discarded. For example, in the Figure 8-2, the subscriber is receiving messages from two destinations simultaneously.

Figure 8-2 Multicasting Sequence Gap



Upon receiving the "4" message from Destination 1, a `SequenceGapException` is thrown to notify the application that a message was received out of sequence. If subsequently received, the "3" message will be discarded.

> **Note**
>
> The larger the messages being exchanged, the greater the risk of encountering a `SequenceGapException`.

Dynamically Configuring Multicasting Configuration Attributes

During configuration, for each connection factory the system administrator configures the following information to support multicasting:

- Message maximum specifying the maximum number of outstanding messages that can exist on a multicast session.
- Overrun policy specifying whether recent or older messages are discarded in the event the message maximum is reached.

When an arriving multicast message would exceed the session message maximum, WebLogic JMS discards a message according to the overrun policy and reports a DataOverrunException to the session's registered ExceptionListener, if any. Applications can configure the message maximum and overrun policy using the WebLogic WLSession extension.

Table 8-1 lists the `WLSession` set and get methods for each dynamically configurable attribute.

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

Table 8-1 WLSession Set and Get Methods

Attribute	Set Method	Get Method
Message Maximum	`public void setMessagesMaximum(int messagesMaximum) throws JMSException`	`public int getMessagesMaximum() throws JMSException`
Overrun Policy	`public void setOverrunPolicy (int overrunPolicy) throws JMSException`	`public int getOverrunPolicy() throws JMSException`

> **Note**
>
> The values set using the set methods take precedence over the configured values.

For more information about these `Session` class methods, see the `weblogic.jms.extensions.WLSession` Javadoc.

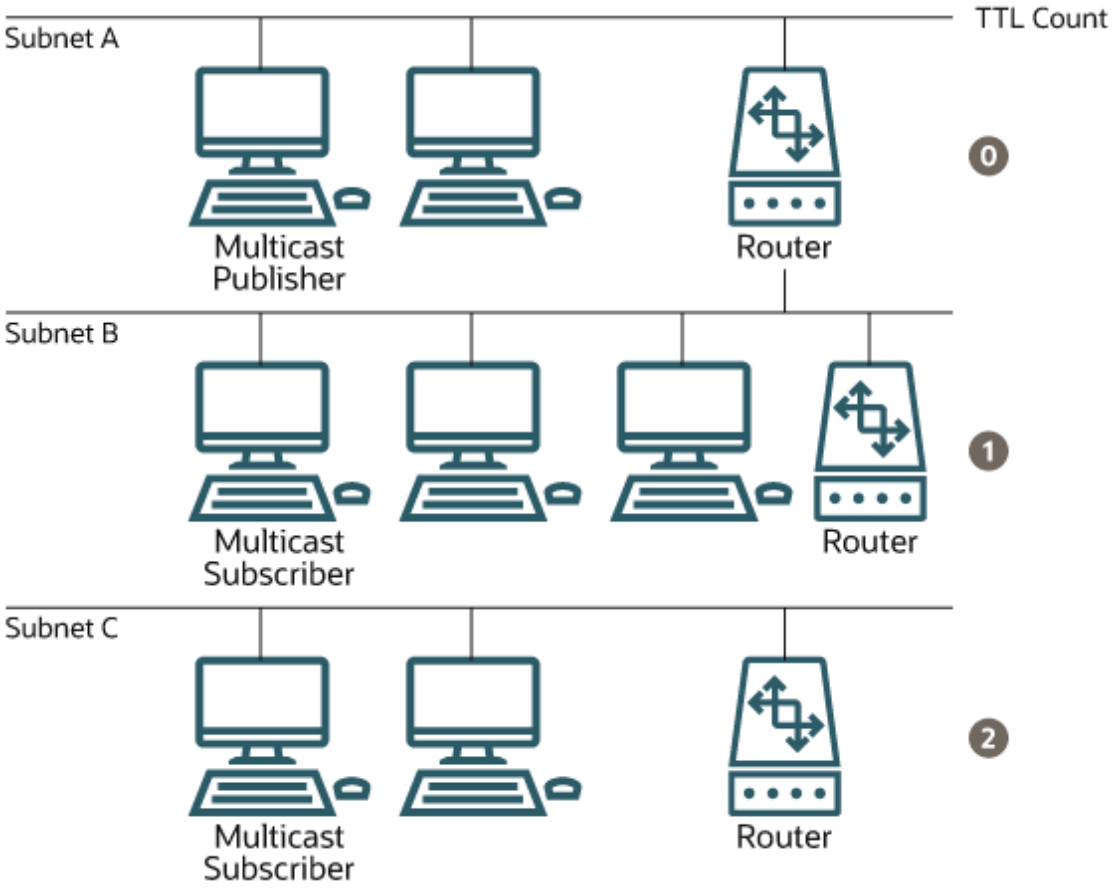
Example: Multicast Time-to-Live

> **Note**
>
> The following example is a very simplified illustration of how the Multicast TTL (time-to-live) destination configuration attribute affects the delivery of messages across routers. It is strongly advised that you seek the assistance of your network administrator when configuring the multicast TTL attribute to ensure that the appropriate value is set.
>
> The Multicast TTL is independent of the message time-to-live.

Figure 8-3 shows how the Multicast TTL destination configuration attribute affects the delivery of messages across routers.

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

Figure 8-3 Multicast TTL Example



In the figure, the network consists of three subnets: Subnet A containing the multicast publisher, and Subnets B and C each contain one multicast subscriber.

If the Multicast TTL attribute is set to 0 (indicating that the messages cannot traverse any routers and are delivered on the current subnet only), when the multicast publisher on Subnet A publishes a message, the message will not be delivered to any of the multicast subscribers.

If the Multicast TTL attribute is set to 1 (indicating that messages can traverse one router), when the multicast publisher on Subnet A publishes a message, the multicast subscriber on Subnet B will receive the message.

Similarly, if the Multicast TTL attribute is set to 2 (indicating that messages can traverse two routers), when the multicast publisher on Subnet A publishes a message, the multicast subscribers on Subnets B and C will receive the message.

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

9

Using Distributed Destinations

Understand the concepts and functionality of distributed destinations necessary to design high availability (HA) applications.

What Is a Distributed Destination?

A distributed destination is a set of destinations (queues or topics) that are accessible as a single, logical destination to a client.

A distributed destination has the following characteristics:

- It is referenced by its own JNDI name.
- Each member of the set can belong to a separate JMS server distributed across multiple servers within a single cluster, or can be located on JMS servers that are all on the same single non-clustered standalone server. Members of the set cannot be distributed across multiple non-clustered standalone servers, and cannot be distributed across multiple clusters.

Why Use a Distributed Destination

Applications that use distributed destinations are more highly available than applications that use simple destinations because WebLogic JMS provides load balancing and failover for member destinations of a distributed destination within a cluster.

Once properly configured, your producers and consumers are able to send and receive messages through the distributed destination. WebLogic JMS then balances the messaging load across all available members of the distributed destination. When one member becomes unavailable due to a server failure, traffic is then redirected toward other available destination members in the set. For more information about how destination members are load balanced, see "Configuring Distributed Destination Resources" in *Administering JMS Resources for Oracle WebLogic Server*.

Creating a Distributed Destination

See Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

Types of Distributed Destinations

Learn about the two types of distributed destinations supported by WebLogic Server.

Uniform Distributed Destinations

In a uniform distributed destination (UDD), each of the member destinations has a consistent configuration of all distributed destination parameters, particularly in regards to weighting, security, persistence, paging, and quotas.

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

Oracle recommends using UDDs because you no longer need to create or designate destination members, but instead rely on WebLogic Server to uniformly create the necessary members on the JMS servers to which a UDD is targeted. This feature of UDDs provides dynamic updating of a UDD when a new member is added or a member is removed.

For example, if a UDD is targeted to a cluster, there is a UDD member on every JMS server in the cluster. If a new JMS server is added, then a new UDD member is dynamically added to the UDD. Likewise, if a JMS server is removed, then the corresponding UDD member is removed from the UDD. This allows UDDs to provide higher availability by eliminating bottlenecks caused by configuration errors. For more information, see Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

Weighted Distributed Destinations

> **Note**
>
> Weighted distributed destinations are deprecated in WebLogic Server 10.3.4.0. Oracle recommends using Uniform Distributed Destinations.

In a weighted distributed destination, the member destinations do not have a consistent configuration of all distributed destination parameters, particularly in regards to weighting, security, persistence, paging, and quotas.

Oracle recommends converting weighted distributed destinations to UDDs because of the administrative inflexibility when creating members that are intended to carry extra message load or have extra capacity (more weight). Lack of a consistent member configuration can lead to unforeseen administrative and application problems because the weighted distributed destination cannot be deployed consistently across a cluster.

For more information, see Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

Using Distributed Destinations

A distributed destination is a set of physical JMS destination members (queues or topics) that is accessed through a single JNDI name.

As such, a distributed destination can be looked up using JNDI. Distributed destination implements the `jakarta.jms.Destination` interface, at `jakarta.jms.Destination`, and can be used to create producers, consumers, and browsers.

For information about obtaining a reference to a distributed destination, see How to Look Up a Destination.

Using Distributed Queues

A distributed queue is a set of physical JMS queue members. As such, a distributed queue can be used to create a *`QueueSender`*, `QueueReceiver`, and a `QueueBrowser`. The fact that a distributed queue represents multiple physical queues is mostly transparent to your application.

The queue members can be located anywhere, but must all be served by JMS servers in a single server cluster. When a message is sent to a distributed queue, it is sent to one of the physical queues in the set of members for the distributed queue. Once the message arrives at the queue member, it is available for receipt by consumers of that queue member only. '

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

This section provides information on using distributed queues:

QueueSenders

After creating a queue sender, if the queue supplied at creation time was a distributed queue, then each time a message is produced using the sender a decision is made as to which queue member will receive the message. Each message is sent to a single physical queue member.

The message is not replicated. As such, the message is only available from the queue member where it was sent. If that physical queue becomes unavailable before a given message is received, then the message is unavailable until that queue member comes back online.

It is not enough to send a message to a distributed queue and expect the message to be received by a queue receiver of that distributed queue. Because the message is sent to only one physical queue member, there must be a queue receiver receiving or listening on that queue member.

> **Note**
>
> For information about the load-balancing heuristics for distributed queues with zero consumers, see Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

QueueReceivers

When creating a queue receiver, if the supplied queue is a distributed queue, then a single physical queue member is chosen for the receiver at creation time. The created `QueueReceiver` is pinned to that queue member until the queue receiver loses its access to the queue member. At that point, the consumer will receive a `JMSException`, as follows:

- If the queue receiver is synchronous, then the exception is returned to the user directly.
- If the queue receiver is asynchronous, then the exception is delivered inside of a `ConsumerClosedException` that is delivered to the `ExceptionListener` defined for the consumer session, if any.

Upon receiving such an exception, an application can close its queue receiver and recreate it. If any other queue members are available within the distributed queue, then the creation will succeed and the new queue receiver will be pinned to one of those queue members. If no other queue member is available, then the application would not be able to recreate the queue receiver and will have to try again later.

> **Note**
>
> For information about the load-balancing heuristics for distributed queues with zero consumers, see Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

QueueBrowsers

When creating a queue browser, if the supplied queue is a distributed queue, then a single physical queue member is chosen for the browser at creation time. The created queue browser is pinned to that queue member until the receiver loses its access to the queue member. At

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

that point, any calls to the queue browser will receive a `JMSException`. Any calls to the enumeration will return a `NoSuchElementException`.

> **Note**
>
> The queue browser can only browse the queue member that it is pinned to. Even though a distributed queue was specified at creation time, the queue browser cannot see or browse messages for the other queue members in the distributed destination.

Using Replicated Distributed Topics

A distributed topic is a set of physical JMS topic members. A distributed topic can be used to create a `TopicPublisher` and `TopicSubscriber`. The fact that a distributed topic represents multiple physical topics is mostly transparent to the application.

> **Note**
>
> Durable subscribers (`DurableTopicSubscriber`) cannot be created for distributed topics. However, you can still create a durable subscription on a distributed topic member and the other topic members will forward the messages to the topic member that has the durable subscription.

The topic members can be located anywhere but must all be served either by a single WebLogic Server or any number of servers in a cluster. When a message is sent to a distributed topic, it is sent to all of the topic members in the distributed topic set. This enables all subscribers to the distributed topic to receive messages published for the distributed topic.

A message published directly to a topic member of a distributed destination (that is, the publisher did not specify the distributed destination) is also forwarded to all the members of that distributed topic. This includes subscribers that originally subscribed to the distributed topic and happened to be assigned to that particular topic member. In other words, publishing a message to a specific distributed topic member automatically forwards it to all the other distributed topic members, just as publishing a message to a distributed topic automatically forwards it to all of its distributed topic members. For more information about looking up specific distributed destination members, see Accessing Distributed Destination Members.

This section provides information on using distributed topics:

TopicPublishers

When creating a topic publisher, if the supplied destination is a distributed destination, then any messages sent to that distributed destination are sent to all available topic members for that distributed topic (DT), as follows:

- When some of the members of a uniform distributed topic are offline, non-persistent messages published to the distributed topic are saved for those members and made available when the members come back online.

  In releases prior to 9.0, if you did not configure a persistent store for a JMS server or if there was a persistent store defined and storedEnabled=false was set on the distributed topic member, non persistent messages were dropped and not made available when the distributed topic member came back online. If your application depends on dropping these

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

messages, you can approximate similar behavior by setting the `time-to-live` for a server to a very low value. This will allow the messages to be disregarded before an offline distributed topic member would come back online. New applications developed on WebLogic Server releases 10.3.4.0 and higher can use partitioned distributed topics with message-driven beans (MDBs) as message consumers to provide a similar capability. See "Developing Advanced Pub/Sub Applications" in Programming JMS for Oracle WebLogic Server.

- If one or more of the distributed topic members is not reachable, and the message being sent is persistent, then the message is stored and forwarded to the other topic members when they become reachable. However, the message can only be persistently stored if the topic member has a JMS store configured.

  > **Note**
  >
  > Every effort is made to first forward the message to distributed members that utilize a persistent store. However, if none of the distributed members utilize a store, then the message is still sent to one of the members according to the selected load-balancing algorithm, as described in Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

- If all of the distributed topic members are unreachable (regardless of whether the message is persistent or non persistent), then the publisher receives a `JMSException` when it tries to send a message.

TopicSubscribers

When creating a topic subscriber, if the supplied topic is a distributed topic, then the topic subscriber receives messages published to that distributed topic. If one or more of the topic members for the distributed topic are not reachable by a topic subscriber, then depending on whether the messages are persistent or non persistent the following occurs:

- Any persistent messages published to one or more unreachable distributed topic members are eventually received by topic subscribers of those topic members after they become reachable. However, the messages can only be persistently stored if the topic member has a JMS store configured.
- Any non persistent messages published to those unreachable distributed topic members will not be received by that topic subscriber.

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

> **Note**
>
> If a JMS store is configured for a JMS server that is hosting a distributed topic member, then all the Distributed Topic System Subscribers associated with that member destination are treated as durable subscriptions, even when a topic member does not have a JMS store explicitly configured. The saving of all the messages sent to these distributed topic subscribers in memory can result in unexpected memory and disk consumption. Therefore, a recommended best design practice when deploying distributed destination is to consistently configure all member destinations: either with a JMS store for durable messages or without a JMS store for non durable messages. For example, if you want all of your distributed topic subscribers to be nondurable, but some member destinations implicitly have a JMS store configured because their associated JMS server uses a JMS store, then you need to explicitly set the `StoreEnabled` attribute to False for each member destination to override the JMS server setting.

Ultimately, a topic subscriber is pinned to a physical topic member. If that topic member becomes unavailable, then the topic subscriber will receive a `JMSException`, as follows:

- If the topic subscriber is synchronous, then the exception is returned to the user directly.
- If the topic subscriber is asynchronous, then the exception is delivered inside of a `ConsumerClosedException` that is delivered to the `ExceptionListener` defined for the consumer session, if any.

After receiving this type of an exception, an application can close its topic subscriber and recreate it. If any other topic member is available within the distributed topic, then the creation should be successful and the new topic subscriber will be pinned to one of those topic members. If no other topic member is available, then the application will not be able to recreate the topic subscriber and will have to try again later.

Deploying Message-Driven Beans on a Distributed Topic

For information about how to deploy MDBs on topics, see Configuring and Deploying MDBs Using Distributed Topics in *Developing Message-Driven Beans for Oracle WebLogic Server*.

Using Partitioned Distributed Topics

Starting in WebLogic Server 10.3.4.0, partitioned distributed topics, combined with the ability to share subscriptions and allow multiple connections to use the same Client ID, provide the following application design patterns that provide parallel processing and HA capabilities similar to distributed queues:

- One-copy-per-instance: Each instance of an application gets one copy of each message that is published to the Topic.
- One-copy-per-application: Each application as a whole (that is all instances of the application together) receives one copy of each message that is published to the distributed topic . That is each instance only receives a subset of the messages that are sent to the distributed topic .

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

> **Note**
>
> Oracle recommends designing applications that utilize WebLogic Server MDBs. See Configuring and Deploying MDBs Using Distributed Topics in *Developing Message-Driven Beans for Oracle WebLogic Server* for detailed information on how to design and implement applications that use message-driven beans to provide improved HA and scalability.

For more information about using Partitioned Distributed Topics, including information about replacing an existing Replicated Distributed Topic with a Partitioned Distributed Topic, see Developing Advanced Pub/Sub Applications.

Accessing Distributed Destination Members

For information on how to access distributed destinations and their members, see How to Look Up a Destination.

Distributed Destination Failover

> **Note**
>
> If the distributed queue member selected for a producer becomes unavailable while the WebLogic Server instance hosting the producer's JMS connection remains running, WebLogic JMS can fail the producer over to another available member, regardless of whether the connection factory's Load Balancing option is enabled. For example, if a producer is connected through WLSServer2 and its selected queue member is hosted on WLSServer1, failure of WLSServer1 can cause subsequent sends to use another available queue member while the connection through WLSServer2 remains active.

A simple way to failover a client connected to a failed distributed destination is to write reconnect logic in the client code to connect to the distributed destination after catching `onException`.

Using Message-Driven Beans with Distributed Destinations

A message-driven bean (MDB) acts as a JMS message listener, which is similar to an event listener except that it receives messages instead of events.

See also:

- MDBs and Messaging Models in *Developing Message-Driven Beans for Oracle WebLogic Server*
- Deploying MDBs in *Developing Message-Driven Beans for Oracle WebLogic Server*

Common Use Cases for Distributed Destinations

Understand when to use distributed destinations for your applications.

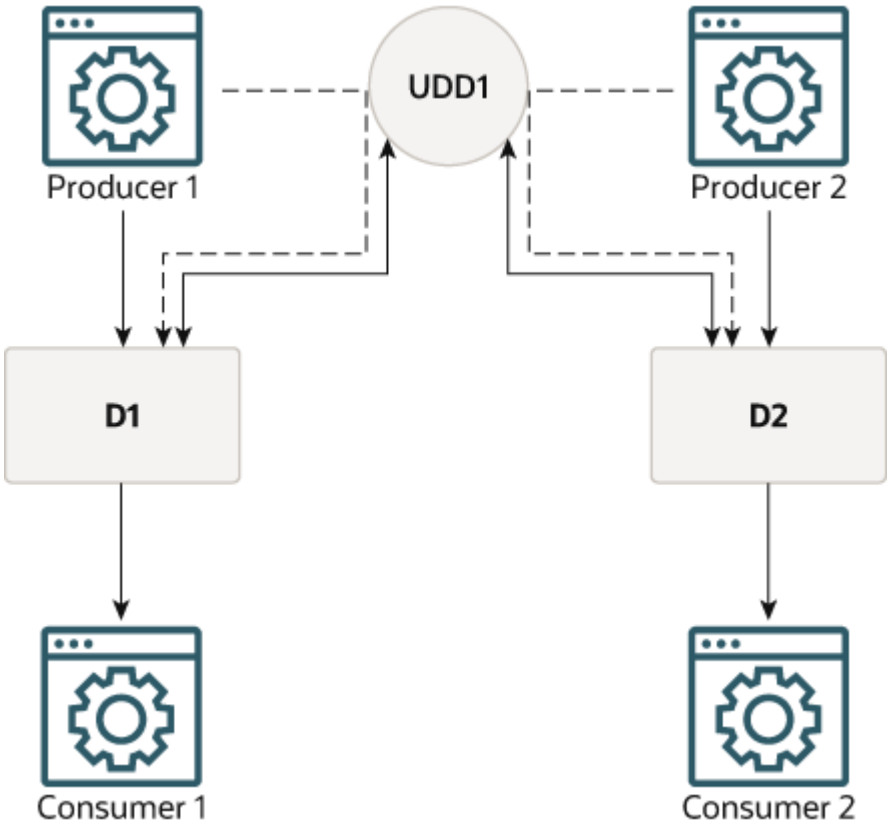
Copyright © 2007, 2026, Oracle and/or its affiliates.

The following sections provide common use case scenarios when using distributed destinations:

Maximizing Production

To maximize message production, Oracle recommends that each member of a distributed destination be associated with a producer and a consumer. Figure 9-1 shows how to efficiently provide maximum message production and high availability using a UDD without using load balancing:

Figure 9-1 Paired Producers and Consumers



In this situation, UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. Each physical destination has a producer/consumer pair and the effective path for a message follows the solid line from the producer through the destination member to the consumer. If you are using ordering, you should have a producer for each expected Unit-of-Order. See Using Unit-of-Order with Distributed Destinations.

Maximizing Availability

This section provides information on how to maximize message availability.

Using Queues

Ideally, it is best to pair a producer with a consumer but it is not always practical. The rate that messages are consumed is the limiting factor that determines the message throughput of your application. You can increase the availability of consumers by using load balancing between

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

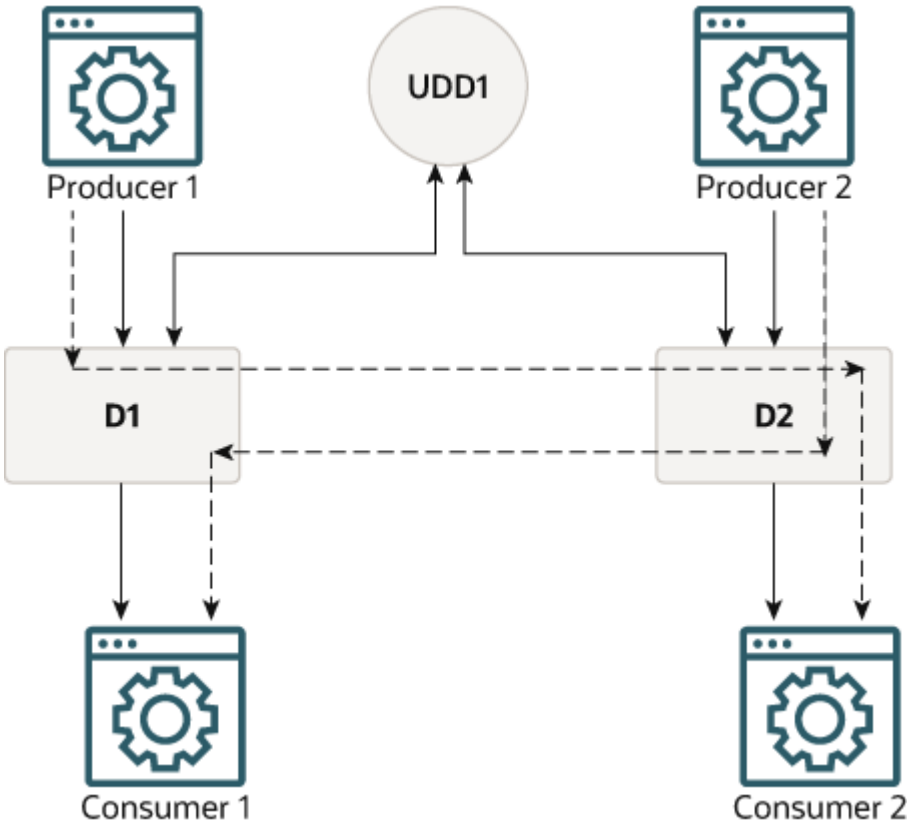
member destinations. In this situation, consumers are not paired with a producer as the UDD load balances an incoming message to the next available consumer using the assigned load balancing algorithm.

> **Note**
>
> Some combinations of Unit-of-Order features can result in the starvation of competing Unit-of-Order message streams, including the under utilization of resources when the number of consumers exceed the number of in-flight messages with different Unit-of-Order names. You will need to test your applications under maximum loads to optimize your system's performance and eliminate conditions that underutilize resources.

Using Topics

For a uniform distributed topic with the Replicated forwarding policy, a message published to one member is forwarded to the other members. Figure 9-2 illustrates this replicated configuration. With the Partitioned forwarding policy, a publication is retained on the receiving member rather than replicated to every member.

Figure 9-2 Using Distributed Topics



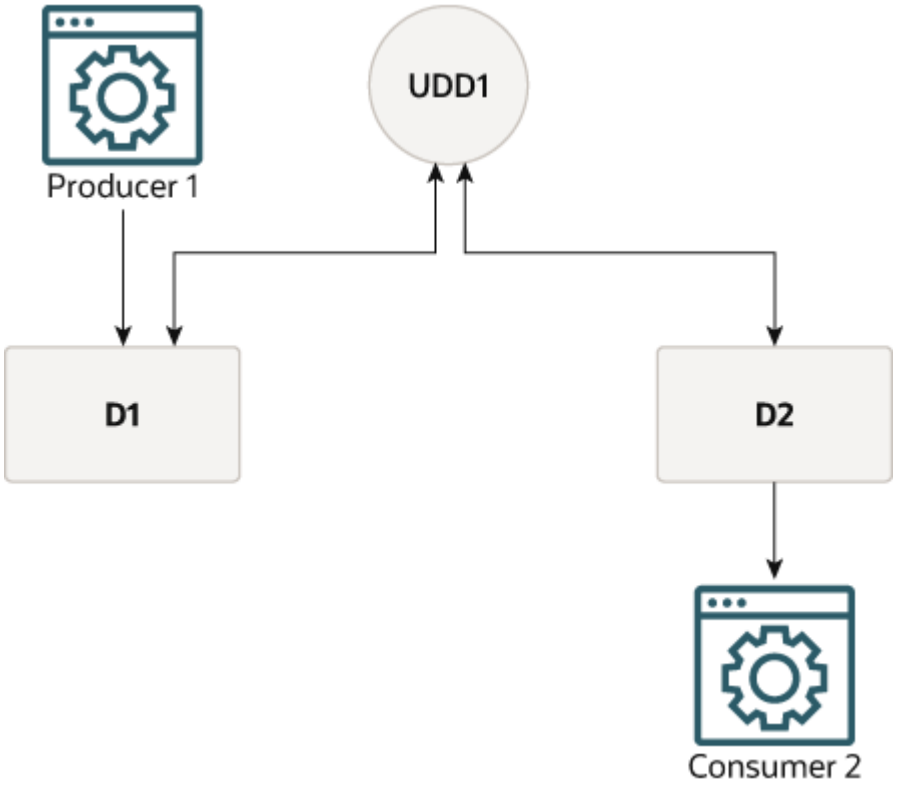
In Figure 9-2, UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. Each physical destination has a producer/consumer pair. Each consumer receives messages sent by Producer 1 and Producer 2.

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

Stuck Messages

InFigure 9-3, a producer is sending messages to one member of a UDD but there is no consumer available to get the message. This typically happens as a producer sends a message to one of the destinations (D1) and a consumer is listening for messages on another destination (D2).

Figure 9-3 Stuck Messages



UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. D1 has a producer and D2 has a consumer. Avoid this configuration by using producer/consumer pairs or by configuring forwarding on the destination.

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

10

Using the Message Unit-of-Order

Learn how to use Message Unit-of-Order to provide strict message ordering when using WebLogic JMS.

What is Message Unit-Of-Order?

Message Unit-of-Order is a WebLogic Server feature that enables a stand-alone message producer, or a group of producers acting as one, to group messages into a single unit with respect to the processing order.

This single unit is called a Unit-of-Order. Messages with the same Unit-of-Order are processed sequentially in the order they are submitted to the destination, according to the Unit-of-Order acknowledgement and transaction rules.

Understanding Message Processing with Unit-of-Order

Understand how the message processing by WebLogic Server's Message Unit-of-Order feature is different from the message processing as described by the JMS specification.

Message Processing According to the JMS Specification

While the Jakarta Messaging 3.1 specification, at Jakarta Messaging 3.1 specification, provides an ordered message delivery, it does so in a very strict sense. It defines order between a single instance of a producer and a single instance of a consumer, but does not take into account the following common situations:

- Many consumers on one queue. See Using Distributed Destinations.
- Multiple producers within a single application acting as a single producer. See Using Distributed Destinations.
- Message recoveries or transaction rollbacks where other messages from the same producer can be delivered to another consumer for processing. See What Happens When a Message Is Delayed During Processing?.
- Use of filters and destination sort keys. See Message Unit-of-Order Advanced Topics.

Message Processing with Unit-of-Order

The WebLogic Server Unit-of-Order feature enables a message producer or group of message producers acting as one, to group messages into a single unit that is called a Unit-of-Order. Messages in the Unit-of-Order are processed sequentially in the order they are submitted to the destination, according to the acknowledgement and transaction rules. The message processing of a single message is complete when a message is acknowledged, committed, recovered, or rolled back. Until message processing for a message is complete, the remaining unprocessed messages for that Unit-of-Order are blocked.

This section provides information about rules for JMS acknowledgement modes, described at Jakarta Messaging 3.1 specification, when using Message Unit-of-Order:

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

- No messages from a Unit-of-Order are processed in parallel when the acknowledgement mode is `CLIENT_ACKNOWLEDGE`, `AUTO_ACKNOWLEDGE`, or `DUPS_OK_ACKNOWLEDGE`.
- When the consumer is closed, the current message processing is completed, regardless of the session's acknowledge mode.
- `CLIENT_ACKNOWLEDGE` – The application calling `Message.acknowledge` and `Session.recover` indicate which messages are completely processed in the Unit-of-Order.
- `AUTO_ACKNOWLEDGE` – The session automatically acknowledges a client's receipt of a message when it has either successfully returned from a call to `receive` or when the `MessageListener` that was called returns successfully.
  - Asynchronous mode: Successful completion or exception of the `onMessage(msg)` indicates when a message is completely processed.
  - Synchronous mode: For a given consumer, such as consumer A, `consumerA.receive` is completed when one of the following occurs: `consumerA.receive`, `consumerA.setMessageListener`, or `consumerA.close`.
- `DUPS_OK_ACKNOWLEDGE` – The session automatically acknowledges a client's receipt of a message when it has either successfully returned from a call to `receive` or when the `MessageListener` that was called returns successfully.
  - Asynchronous mode: Successful completion or exception of `onMessage(msg)` indicates when a message is completely processed.
  - Synchronous mode: For a given consumer, such as consumer A, `consumerA.receive()` is completed when one of the following occurs: `consumerA.receive()`, `consumerA.setMessageListener()`, or `consumerA.close()`.
- `NO_ACKNOWLEDGE` – The session provides no order processing guarantees. Messages can be processed in parallel with different available consumers.

Message Delivery with Unit-of-Order

Message Unit-of-Order provides that messages are delivered in accordance with the following rules:

- Member messages of a Unit-of-Order are delivered to queue consumers sequentially in the order they are submitted to the destination. The message order within a Unit-of-Order will not be affected by sort criteria, priority, or filters. However, messages that are uncommitted, have a `Redelivery Delay`, or have an unexpired `TimetoDeliver` timer will delay messages that arrive after them.
- Unit-of-Order messages are processed one at a time. The processing completion of one message allows the next message in the Unit-of-Order to be delivered.
- Unit-of-Order messages sent to a distributed queue reside on only one physical member of the distributed queue. For more information, see Using Unit-of-Order with Distributed Destinations.
- All uncommitted or unacknowledged messages from the same Unit-of-Order must be in the same transaction, or if non-transactional, the same `JMSSession`. When one message in the Unit-of-Order is uncommitted or unacknowledged, the other messages are deliverable only to the same transaction or `JMSSession`. This keeps all unacknowledged messages from the same Unit-of-Order in one recoverable operation and allows order to be maintained despite rollbacks or recoveries.
- The following examples show how rollback, recovery, and acknowledgement affect delivery of the next message in a Unit-of-Order.

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

For example:

- when Messages M_1 through M_n are delivered as part of a transaction and the transaction is rolled back (processing is complete). Then messages M_1 through M_n are delivered to any available consumer:
- when Messages M_1 through M_n are delivered outside of a transaction and the messages are recovered (processing is complete). Then messages M_1 through M_n are delivered to any available consumer.
- when Messages M_1 through M_n are delivered outside of a transaction and the messages are acknowledged (processing is complete). Then the undelivered message M_{n+1} is delivered to any available consumer.

Message Unit-of-Order Case Study

Learn the features of Message Unit-of-Order through a case study based on ordering a book from an online bookstore.

Joe Orders a Book

XYZ Online Bookstore implements a simple processing design that uses JMS to process customer orders. The JMS processing system is composed of:

- A message producer sending to a queue (Queue1).
- Multiple message driven beans (MDBs), such as MdbX and MdbY, that process messages from Queue1.
- A database (myDB) that contains order and order status information.

Joe logs in his XYZ Online Bookstore account and searches his favorite book topics. He chooses a book, proceeds to the checkout, and completes the sales transaction. Then Joe realizes he has previously purchased this same item, so he cancels the order. One week later, the book is delivered to Joe.

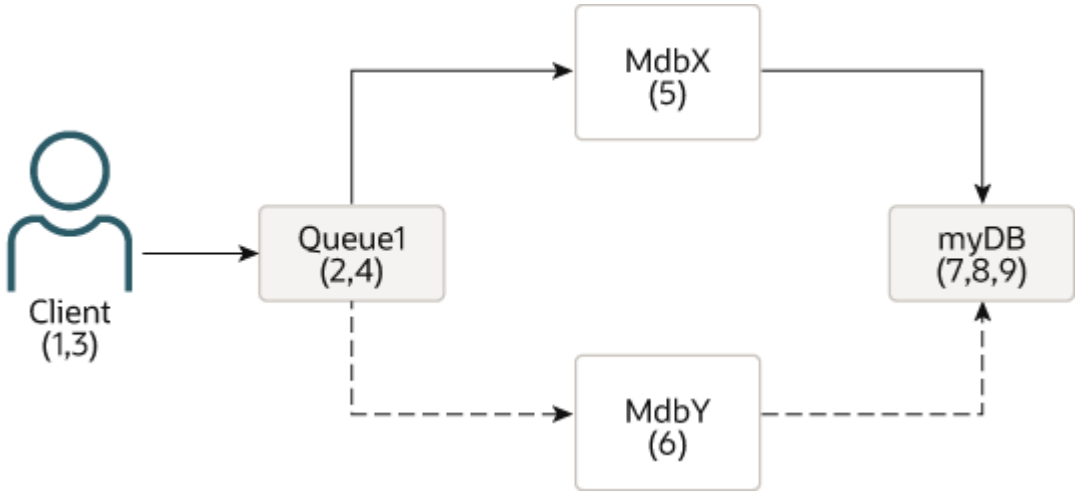
What Happened to Joe's Order

In Joe's ordering scenario, his cancel order message was processed before his purchase order message. The result was that Joe received a book he did not wish to purchase. The following steps demonstrate how Joe's order was processed.

The Figure 10-1 and the corresponding actions demonstrate how Joe's order was processed.

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

Figure 10-1 Workflow for Joe's Order



1. Joe clicks the order button from his shopping cart.
2. The order message (message A) is placed on **Queue1**.
3. Joe cancels the order.
4. The cancel order (message B) is placed on **Queue1**.
5. **MdbX** takes message A from **Queue1**.
6. **MdbY** takes message B from **Queue1**.
7. **MdbY** writes the cancel message to the database. Because there is no corresponding order message, there is no order message to remove from the database.
8. **MdbX** writes the order message to the database.
9. An application responsible for shipping books reads the database, sees the order message, and initiates shipment to Joe's home.

Although the Jakarta Messaging 3.1 specification, at Jakarta Messaging 3.1 specification, provides an ordered message delivery, it only provides ordered message delivery between a single instance of a producer and a single instance of a consumer. In Joe's case, multiple MDBs were available to consume messages from Queue1 and the processing order of the messages was no longer guaranteed.

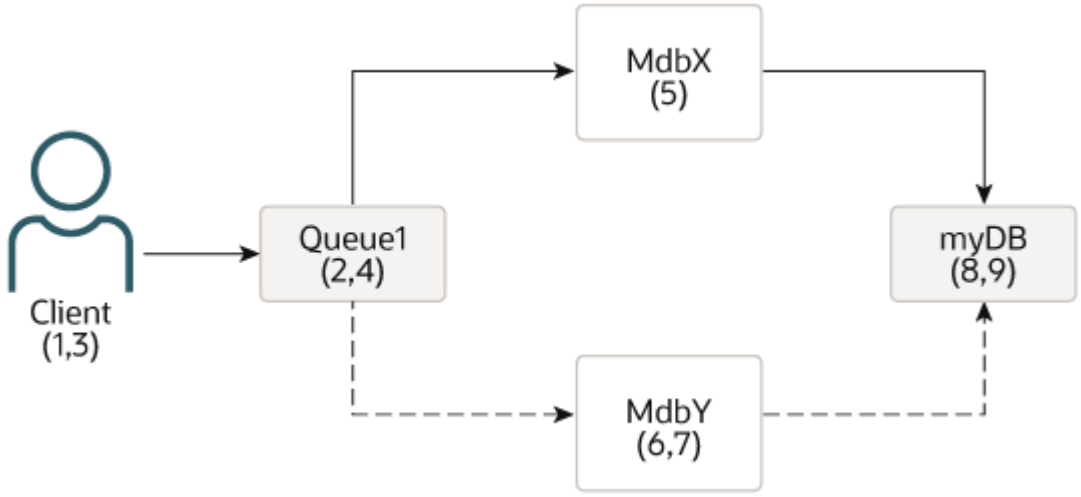
How Message Unit-of-Order Solves the Problem

To ensure that all messages in Joe's order are processed correctly, the system administrator for XYZ Bookstore configures a Message Unit-of-Order based on a user session, such that all messages from a user session have a Unit-of-Order name attribute with the value of the session id. See How to Create a Unit-of-Order. Messages that Joe's application submits to the destination with that Unit-of-Order are processed sequentially in submission order. Applications must submit messages in the intended order; creating a message object earlier does not cause it to be processed first.

In Figure 10-2 and the corresponding actions demonstrate how Joe's order was processed using Message Unit-of-Order.

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

Figure 10-2 Workflow for Joe's Order Using Unit-of-Order



1. Joe clicks the order button from his shopping cart.
2. The order message (message A) is placed on **Queue1**.
3. Joe cancels the order.
4. The cancel order (message B) is placed on **Queue1**.
5. **MdbX** takes message A from Queue1.
6. **MdbY** takes message B from Queue1.
7. Message B on **MdbY** is blocked until **MdbX** acknowledges the order message. See What Happens When a Message Is Delayed During Processing?.
8. Message A is committed and written to the database.
9. Message B is committed and written to the database.

   Because there is a corresponding order message, Joe's order is removed from the database and he does not receive a book.

How to Create a Unit-of-Order

Learn how to create a Message Unit-of-Order programmatically and administratively.

Also see Message Delivery with Unit-of-Order and Message Unit-of-Order Advanced Topics.

Creating a Unit-of-Order Programmatically

Use the `setUnitOfOrder()` method of the `WLMessageProducer` interface to associate a producer with a Unit-of-Order name.

In the following example, the Unit-of-Order name attribute value is set to *myUOOname*:

```
getProducer().setUnitOfOrder("myUOOname");
```

After a producer is associated with a Unit-of-Order, all messages sent by this producer are processed as a Unit-of-Order until either the producer is closed or the association between the producer and the Unit-of-Order is dissolved.

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

The Example 10-1 shows how to associate a producer with a Unit-of-Order:

Example 10-1 Using the WLMessageProducer Interface to Create a Unit-of-Order

```
.
.
.
queue = (Queue)(ctx.lookup(destName));
qsender = (WLMessageProducer) qs.createProducer(queue);
qsender.setUnitOfOrder();
uooname = qsender.getUnitOfOrder();
System.out.println("Using UnitOfOrder :" + uooname);
.
.
.
```

Unit-of-Order Naming Rules

A Unit-of-Order is identified by a name attribute. Within a destination, messages that have the same value for the Unit-of-Order name attribute belong to the same Unit-of-Order. The name can be provided by either the system or the application. Messages in the same Unit-of-Order all share the same name. See How to Create a Unit-of-Order.

The name attribute for a Unit-of-Order must adhere to the following rules:

- A valid value for the Unit-of-Order name attribute is any non-null and non-empty string.
- System-generated Unit-of-Order names are timestamp-based and statistically unique.
- Applications can supply their own Unit-of-Order names. For example, WebLogic Integration applications can use Workflow names and Web Services applications can use conversation names.
- Message Unit-of-Order has its own name space. A Unit-of-Order does not need to be unique with respect to other named objects. For instance, it is valid to have a Unit-of-Order named *Foo* and a queue named *Foo*.
- The scope of a Message Unit-of-Order is limited to a single destination. Two different Units of Order on two destinations can have the same name.
- One or more producers can send messages with the same Unit-of-Order name by using the same string to create the Unit-of-Order.

  A system-generated Unit-of-Order name can be used by more than one producer. This paradigm works just as well for application-assigned Unit-of-Order names. It will be most efficient if the information is serialized in only one place, so a property like Conversation ID can be stored only as the Unit-of-Order name. This paradigm does not work when the message is sent through a non-Unit-of-Order JMS provider (releases before WebLogic 9.0 or non-WebLogic JMS providers).

Getting the Current Unit-of-Order

You can extract the Unit-of-Order name from a delivered message.

For example:

```
msg.getStringProperty("JMS_BEA_UnitOfOrder");
```


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

Message Unit-of-Order Advanced Topics

Understand how Unit-of-Order processes messages in advanced or more complex situations.

What Happens When a Message Is Delayed During Processing?

There are many situations that can occur during message processing that would normally change the order in which a message is processed. The following is a short list of typical message processing states that make a message not ready for delivery:

- A message is within an uncommitted transaction.
- A message's `TimeToDeliver` value prevents it from being delivered until the `TimeToDeliver` interval has elapsed.
- A consumer calls a recover or rollback operation that prevents a message from being re-delivered until the `RedeliveryDelay` interval has elapsed.

Suppose messages A and B arrive respectively in the same Unit-of-Order, and message A cannot be delivered for any of the previously listed reason. Even though nothing is delaying the delivery of message B, it is not deliverable until message A in its Unit-of-Order is delivered.

What Happens When a Filter Makes a Message Undeliverable

Using a filter and a Unit-of-Order can provide unexpected behaviors. Suppose messages A through Z are in the same Unit-of-Order in the same Queue. Consumer1 has a filter, and messages A, B, and C satisfy the filter, and they are delivered to Consumer1.

1. Messages D through Z are undeliverable until messages A, B, and C are acknowledged.
2. Messages A, B, and C are acknowledged or recovered.
3. Message D is available to the message delivery system.
4. Message D does not pass the filter and can never be presented to Consumer1.
5. Messages E through Z are undeliverable until message D is processed.

- The transaction that contains message D must be rolled back.
- After message D is processed, messages E through Z can be delivered.

For more information, see Filtering Messages.

What Happens When Destination Sort Keys Are Used

Destination sort keys control the order in which messages are presented to consumers when messages are not part of a Unit-of-Order or are not part of the same Unit-of-Order.

For example, messages A and B arrive in the same Unit-of-Order on a queue that is sorted by priority and the sort order depends, but message B has a higher priority than A.

Even though message B has a higher priority than message A, message B is still not deliverable until message A is processed because they are in the same Unit-of-Order. If a message C arrives and either does not have a Unit-of-Order or is not in the same Unit-of-Order as message A, then the priority setting of message C and the priority setting of message A determine the delivery order. See Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*.

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

Using Unit-of-Order with Distributed Destinations

As previously discussed in the Message Processing According to the JMS Specification, the Jakarta Messaging 3.1 specification (at Jakarta Messaging 3.1 specification) does not guarantee ordered message delivery when applications use distributed queues. WebLogic JMS directs messages with the same Unit-of-Order and having a distributed destination target to the same distributed destination member. The member is selected by the destination's Unit-of-Order configuration:

Using the Path Service

You can configure the WebLogic Path Service to provide a persistent map that can store the information required to route the messages contained in a Unit-of-Order to its destination resource; a member of a uniform distributed destination. If the WebLogic Path Service is configured for a uniform distributed destination, then the routing path to a member destination is determined by the server using the run-time load balancing heuristics for the distributed queue.

Using Hash-Based Routing

If the WebLogic Path Service is not configured, then the default routing path to a uniform queue member is chosen by the server based on the hash codes of the Message Unit-of-Order name and the uniform distributed queue members. An advantage of this routing mechanism is that routes to a distributed queue member are calculated quickly and do not require persistent storage in a cluster.

Consider the following when implementing Message Unit-of-Order in conjunction with hash-based routing:

- If a distributed queue member has an associated Unit-of-Order and is removed from the distributed queue, new messages are sent to a different distributed queue member and the messages will not be continuous with older messages.
- If a distributed Queue member has an associated Unit-of-Order and is unreachable, then the producer sending the message will throw a `JMSOrderException` and the messages are not routed to other distributed Queue members. The exception is thrown because the JMS messaging system cannot meet the required quality of service only one distributed destination member consumes messages for a particular Unit-of-Order.

Using Unit-of-Order with Topics

Assigning a Unit-of-Order does not prohibit parallel processing of a message by two subscribers on the same topic. Because individual subscribers for a topic have their own destination and message list, similar to a queue with one consumer, messages are processed by all subscribers according to the Unit-of-Order assigned at the time of production.

Unit-of-Order and Distributed Topics

The routing of messages between physical topics can affect Unit-of-Order if an application directly sends to a member of a distributed topic. To ensure correct order of processing, the application must ensure the messages are sent using the logical distributed topic (that is, the destination is obtained using the JNDI name of the distributed topic). WebLogic Server then ensures messages with the same Unit-of-Order take the same path to the distributed topic member.

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

Unit-of-Order, Topics, and Message Driven Beans

The WebLogic Server message-driven bean implementation goes beyond the requirements of the EJB and JMS specifications to provide parallel processing of an incoming message stream for a single topic subscription and JMS session. This parallel processing does not take Unit-of-Order into account by default for the non-transactional case, so care is required to ensure that the processing is still ordered correctly.

When using Unit-of-Order with topics and non-transactional message driven beans, apply one or more of the following to ensure that the UOO ordering contract is honored:

- Use a Non-Default Message Distribution Policy: Set the Topic Message Distribution Policy to either One Copy Per Server or One Copy Per Application. See Setting Message Distribution Tuning in *Developing Message-Driven Beans for Oracle WebLogic Server*.
- Enable Container-Managed Transactions: See Configuring Transaction Management Strategy for an MDB in *Developing Message-Driven Beans for Oracle WebLogic Server*. If performance is a concern, you may want to consider additionally enabling transaction batching, which can mitigate the performance impact and often yields better performance than the non-transactional case. See Use Transaction Batching in *Tuning Performance of Oracle WebLogic Server*.
- Set Pool Size to One: Setting the pool size to one has a drastic effect on parallelism. See the description for `max-beans-in-free-pool` element in Deployment Elements and Annotations for MDBs in *Developing Message-Driven Beans for Oracle WebLogic Server*.

Using Unit-of-Order with JMS Message Management

JMS message management allows a JMS administrator to move and delete most messages in a running JMS Server. This enables an administrator to violate the delivery rules specified in Message Delivery with Unit-of-Order.

If messages A, B, C, and D are produced and sent to destination D1 and belong to Unit-of-Order *foo*, consider the following:

- Moving messages C and D to destination D2 may allow parallel processing of messages from both destinations.
- Moving messages B and C to destination D2 may allow parallel processing of message A and messages B and C. After message A is processed, message D is deliverable.

For applications that depend on maintaining message order, a best practice is to move all of the messages in a Unit-of-Order as a single group.

To ensure Unit-of-Order delivery rules are maintained, use the following steps:

1. Pause the source destination and the target destination.
2. Select all of the messages with the Unit-of-Order you would like to move.
3. Move the selected messages to the target destination. If necessary, sort them according to the order that you want them processed.
4. Resume the source and target destinations.

For more information, see "Troubleshooting WebLogic JMS" in *Administering JMS Resources for Oracle WebLogic Server*.

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

Using Unit-of-Order with WebLogic Store-and-Forward

WebLogic Store-and-Forward supports Message Unit-of-Order. For example, a Store-and-Forward producer sends messages with a Unit-of-Order named *Foo*. If the producer disconnects and reconnects through a different connection, the producer creates another Unit-of-Order with the name *Foo* and continues sending messages. All messages sent before and after the reconnect are directed through the same Store-and-Forward agent. See *Administering the Store-and-Forward Service for Oracle WebLogic Server*.

Using Unit-of-Order with WebLogic Messaging Bridge

If both the source and target destinations are WebLogic Server 9.0 or later Messaging Bridge instances, you can enable `PreserveMsgProperty` on the Messaging Bridge to preserve the Unit-of Order name and set the producer's Unit-of-Order accordingly. See *Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.

Limitations of Message Unit-of-Order

Understand the limitations when using Message Unit-of-Order.

- A browser enumeration contains the current queue messages in the order they are to be received by the browser, where *current* is defined as those messages that are deliverable. At most, the first message within a Unit-of-Order is deliverable. Subsequent messages in the same Unit-of-Order are not deliverable.
- Some combinations of Unit-of-Order features can result in the starvation of competing Unit-of-Order message streams, including the underutilization of resources when the number of consumers exceeds the number of in-flight messages with different Unit-of-Order names. You will need to test your applications under maximum loads to optimize your system's performance and eliminate conditions that underutilize resources.
- This release of WebLogic Server Message Unit-of-Order does not support clients connecting to a non-Unit-of-Order JMS provider (releases earlier than WebLogic Server 9.0 or non-WebLogic JMS providers).

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

11

Using Unit-of-Work Message Groups

Learn how to use Unit-of-Work Message Groups to provide groups of messages when using WebLogic JMS.

What Are Unit-of-Work Message Groups?

WebLogic JMS provides the Unit-of-Work (UOW) Message Groups, which allows applications to send JMS messages, identifying some of them as a group and allowing a JMS consumer to process them as such.

The Unit-of-Work (UOW) Message Groups can be used when applications need an even more restricted notion of a group than provided by the Message Unit-of-Order (UOO) feature. For example, a JMS producer can designate a set of messages that must be delivered to a single client without interruption, so that the messages can be processed as a unit. Further, the client will not be blocked waiting for the completion of one unit when there is another unit that is already complete.

> **Note**
>
> It is a programming error to use both the Unit-of-Order and Unit-of-Work features on the same JMS message.

The following sections describe how to use Message UOW to provide strict message grouping when using WebLogic JMS:

- Understanding Message Processing with Unit-of-Work
- How to Create a Unit-of-Work Message Group
- Message Unit-of-Work Advanced Topics
- Limitations of UOW Message Groups

Understanding Message Processing with Unit-of-Work

Understand the basic UOW terminology and the rules for processing UOW messages.

Basic UOW Terminology

Table 11-1 defines the terms used to define UOW.

Table 11-1 Unit-of-Work Terminology

Term	Definition
Unit-of-Work (UOW)	A set of JMS messages that must be processed as a single unit.

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

Table 11-1 (Cont.) Unit-of-Work Terminology

Term	Definition
UOW Component Message	A message that is part of a UOW. In order for WebLogic JMS to identify a message as part of a UOW, the message must have the JMS properties described in How to Write a Producer to Set UOW Message Properties.
UOW Producer	A producer that needs to split its work into multiple parts (i.e., a creator of a UOW). Multiple producers can concurrently contribute component messages to a UOW message, as shown in Message Unit-of-Work Case Study. In fact, a UOW producer can close midway through a UOW and a new producer can complete the UOW message, while maintaining the same strict component message integrity (that is, detect duplicates, etc.).
Intermediate Destination	A destination whose consumers have the job of processing component messages separately rather than as a unit. No special UOW configuration is required for intermediate destinations. When a component message arrives on an intermediate destination it will be made available without waiting for other component messages to arrive. Further, if the intermediate destination is a distributed destination, no special routing needs to occur. See How to Write a UOW Consumer/Producer For an Intermediate Destination.
Terminal Destination	A destination whose consumers have the job of processing a full UOW. A destination is identified as a *terminal destination* by the Unit-of-Work Message Handling Policy parameter on standalone destinations, distributed destinations, or JMS templates. See Configuring Terminal Destinations.
Available/Visible Messages	Equivalent JMS terms that refer to a message becoming ready for consumption, pending the reception of any messages that precede it. For example, a JMS message is not available until its time to deliver has been reached or a JMS message that is sent as part of a transaction is not visible until that transaction is committed.

Rules For Processing UOW Messages

The following rules apply to UOW messages.

- **All Messages Required For Processing**

  No message within the UOW will be available until all of them are available on the terminal destination.

- **Message Reordering**

  No matter what order the messages arrive to the terminal destination, they will be put into the order specified by the UOW producer.

- **Gap Freedom**

  The group of messages will be delivered to the user without gaps. That is, all messages in the group will be delivered to the user before messages from any other group (or part of no group at all).

- **Single Consumer Consumption**

  The group of messages will be delivered to the same consumer.

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

Message Unit-of-Work Case Study

This section provides a simple case study for Message Unit-of-Work based on an online order that requires a variety of merchandise from multiple companies.

Jill Orders Miscellaneous Items from an Online Retailer:

The *Megazon* online retailer implements a processing design that uses JMS to process customer orders for a variety of merchandise, some of which need to be routed to Megazon's partner companies to complete the order. For example, Megazon can directly fulfill book orders, but must reroute some parts of the order for certain electronics or houseware items. Since Megazon is configured to use UOW, items in an order can be routed as UOW messages to these intermediate company destinations before being passed onto Megazon's terminal destination where all the UOW messages that make up the order are gathered before a final invoice can be processed.

The Megazon JMS processing system is composed of:

- A UOW producer sending order fulfillment component messages with the required UOW properties to the appropriate intermediate and terminal destinations
- Intermediate destinations for items other than books, where UOW component messages are processed by consumer and/or producer clients before being passed onto the final UOW destination
- A UOW terminal destination where the component messages are gathered for final processing

Jill logs into her Megazon account and does some holiday shopping. She chooses a book, flash drive, MP3 player, and a lava lamp. She then proceeds to the checkout, and completes the sales transaction.

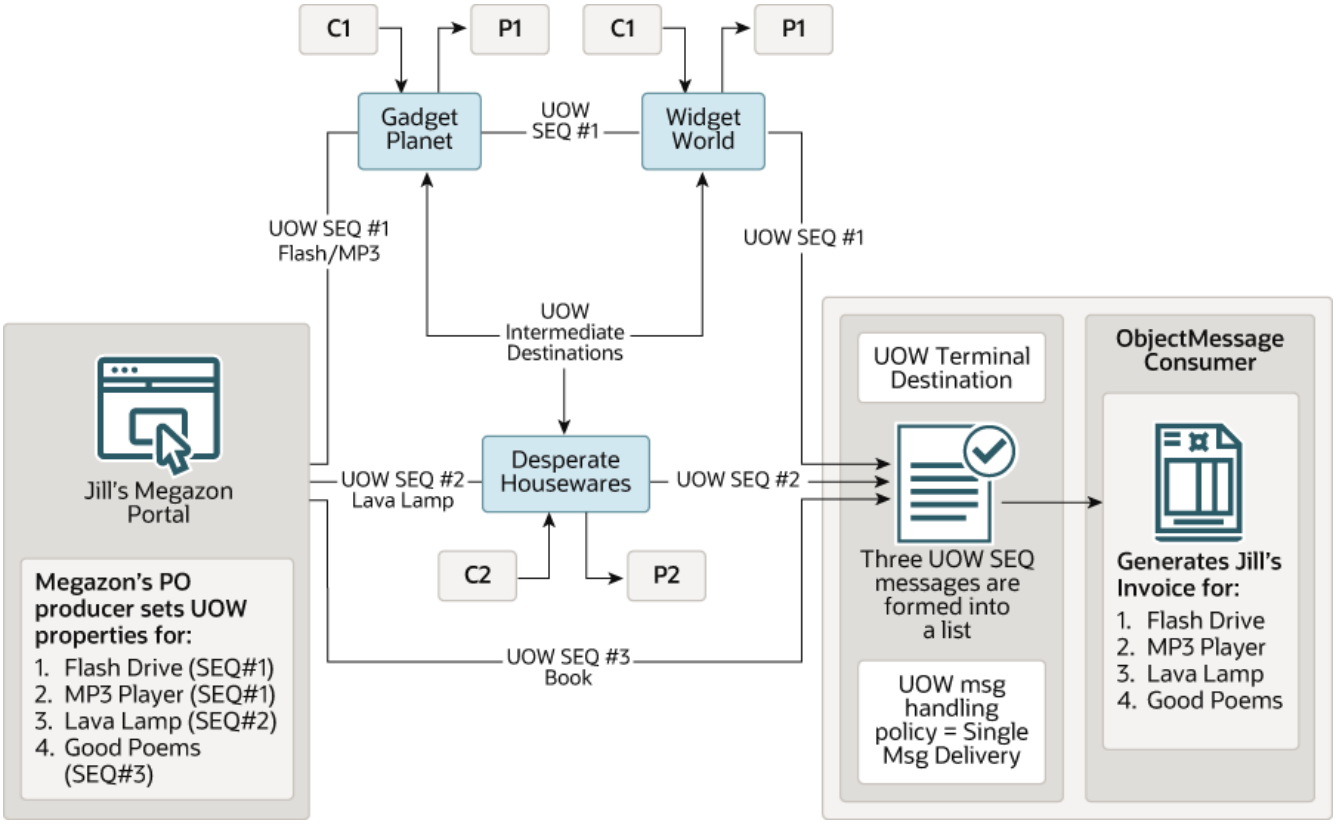
How Message Unit-of-Work Completes the Order:

To ensure that all messages in Jill's order are processed as a single unit, the order-taking JMS producer client sets UOW properties on her order messages to indicate that they are part of a single unit. These UOW message properties must also be copied by any consumer or producer clients listening on the intermediate Gadget Planet, Widget World, and Desperate Housewares destinations before they pass the UOW messages onto the terminal destination. Finally, the system administrator sets the terminal destination’s Unit-of-Work Message Handling Policy to Single Message Delivery. See How to Create a Unit-of-Work Message Group.

Figure 11-1 and the corresponding actions demonstrate how Jill's order was processed using Message Unit-of-Work.

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

Figure 11-1 Workflow for Jill's Order Using Unit-of-Work



1. Jill clicks the order button from her shopping cart.
2. The order is split into three messages that use the same unique UOW name:
   - SEQ#1, which is routed to the intermediate Gadget Planet queue, where a consumer processes the Flash Drive order before passing SEQ#1 onto a producer who then routes it to the intermediate Widget World queue, where a consumer processes the MP3 player order before passing SEQ#1 to the terminal Megazon queue for final invoice processing.
   - SEQ#2, which is routed to the intermediate Desperate Housewares queue, where a consumer processes the lava lamp order before passing SEQ#2 onto a producer who routes it to the Megazon terminal processing queue for final invoice processing.
   - SEQ#3, which is routed directly to Megazon's terminal queue for book order fulfillment and for final invoice processing.
3. The terminal Megazon queue gathers the three UOW messages before forming them into an `ObjectMessage` list for delivery to Megazon's invoice consumer client.
4. Jill receives an invoice that shows her entire order was processed.

How to Create a Unit-of-Work Message Group

Learn how to set UOW message properties for a message consumer and producer.

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

How to Write a Producer to Set UOW Message Properties

UOW enables a producer to split its work into multiple parts to accomplish its goal. UOW is, in effect, taking these multiple messages and joining them into one. Whether component messages are delivered as parts of a single message or as many messages, it is easiest to envision them as a single virtual message, as well as individual messages.

In order for WebLogic JMS to identify a message as part of a UOW, the message must have the JMS properties in Table 11-2 set by the producer client.

Table 11-2 Unit-of-Work Properties

Type	Description
`JMS_BEA_UnitOfWork`	A string property that is set by the standard JMS mechanism for setting properties. For example: `message.setStringProperty("JMS_BEA_UnitOfWork", "MyUnitOfWorkName")` To avoid naming conflicts, the UOW ID should never be reused. For example, if messages are lost or retransmitted, then they may be perceived as part of a separate UOW. For this reason, Oracle recommends using a Java universally unique identifier (UUID). See `http://docs.oracle.com/javase/8/docs/api/java/util/UUID.html`.
`JMS_BEA_UnitOfWorkSequenceNumber`	An integer property that is set by the standard JMS mechanism for setting properties. For example: `message.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber", 5)` The valid values are integers greater than or equal to 1. At a terminal destination, sequence numbers are also constrained by MaximumUnitOfWorkMessages. Number the components consecutively from 1 and mark the final component.
`JMS_BEA_IsUnitOfWorkEnd`	A Boolean property that is set by the standard JMS mechanism for setting properties. For example: `message.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd", true)` When this property is set to true, the message is the last in the Unit-of-Work. When this property is false or nonexistent, the message is not last in the Unit-of-Work.

If the `UnitOfWork` property is not set, then `SequenceNumber` and `End` will be ignored.

Example UOW Producer Code

The Example 11-2 copies the UOW properties defined in Table 11-2.

Example 11-1 Sample UOW Producer Message Properties

This fragment uses an existing writable jakarta.jms.Message sendMsg and configured QueueSender qSender in a method that handles or declares jakarta.jms.JMSException. The 100 components fit the default maximum of 200.

```
String workName = java.util.UUID.randomUUID().toString();
for (int i = 1; i <= 100; i++) {
    sendMsg.setStringProperty("JMS_BEA_UnitOfWork", workName);
    sendMsg.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber", i);
```


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

```
        sendMsg.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd", i == 100);
        qSender.send(sendMsg, jakarta.jms.DeliveryMode.PERSISTENT, 7, 0);
    }
```

UOW Exceptions

A send to a terminal destination can fail when a component violates the UOW sequencing rules. The implementation uses BadSequenceNumberException, OutOfSequenceRangeException, and DuplicateSequenceNumberException in weblogic.jms.common for the conditions below. These are internal implementation classes; application error handling should use jakarta.jms.JMSException rather than depend on those internal types.

- `BadSequenceNumberException` – This will occur if (a) `UnitOfWork` is set on the message, but `SequenceNumber` is not or (b) the `SequenceNumber` is less than or equal to zero.
- `OutOfSequenceRangeException` – This will be thrown if (a) a message is sent with a SequenceNumber that is higher than the sequence number of the message which has already been marked as the end of the unit or (b) a message is sent with a sequence number which is lower than a message which has already arrived in the same unit, yet the new message is marked as the end message. This error can also occur when an incomplete UOW exceeds the configured maximum; after a decrease below an already accepted component number, further components are rejected until the maximum is raised or the work expires or is removed. An already accepted end component retains its established range.
- `DuplicateSequenceNumberException` – This will be thrown to the producer if it sends a message with a sequence number which is the same as a previously sent message in the same UOW.
- `JMSException` – A JMS exception will be thrown if a message has both the `UnitOfOrder` property set and the `UnitOfWork` property set.

> **Note**
>
> As a programming best-practice, consider having your UOW producers send all component messages that comprise a new UOW under a single transaction. This way, either all of the work is completed or none of it is. For example, if a UOW producer gets an exception or crashes partway through a UOW and wants to then cancel the current UOW, then the entire transaction will be rolled back and the application will not need to make a decision for each message after a failure.

How to Write a UOW Consumer/Producer For an Intermediate Destination

An *intermediate destination* is one whose consumers have the job of processing component messages separately rather than as a unit. A JMS `ForwardHelper` extension API is available to assist developers who are writing producers and/or consumers at intermediate destinations. This is because there are many message properties that need to be copied from the incoming message to the outgoing message. For example, the message properties that control the behavior of UOW need to be copied.

The following intermediate consumer code sample copies the UOW properties defined in Table 11-2.

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

Example 11-2 Sample Client Code for UOW Intermediate Destination

This fragment assumes qReceiver1, qsess and qsend are configured JMS objects, incoming messages are TextMessage components, and the enclosing method handles or declares jakarta.jms.JMSException. Existing transaction completion belongs to the calling application.

```
jakarta.jms.Message received = qReceiver1.receive();
if (received != null) {
    jakarta.jms.TextMessage component = (jakarta.jms.TextMessage) received;
    jakarta.jms.TextMessage forwarded = qsess.createTextMessage(component.getText());
    forwarded.setStringProperty("JMS_BEA_UnitOfWork",
        component.getStringProperty("JMS_BEA_UnitOfWork"));
    forwarded.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber",
        component.getIntProperty("JMS_BEA_UnitOfWorkSequenceNumber"));
    if (component.propertyExists("JMS_BEA_IsUnitOfWorkEnd")) {
        forwarded.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd",
            component.getBooleanProperty("JMS_BEA_IsUnitOfWorkEnd"));
    }
    qsend.send(forwarded);
}
```

Note that the three UOW properties are copied from the incoming message to the outgoing message.

Configuring Terminal Destinations

A destination is identified as a *terminal destination* by the Unit-of-Work Message Handling Policy parameter on standalone destinations, distributed destinations, or JMS templates. There is also a parameter that allows for expiration of incomplete work on terminal destinations.

The Advanced configuration options for all destination types are available by using the `DestinationBean` API, as well as on JMS templates by using the `TemplateBean` API.

Table 11-3 Unit-of-Work Configuration Options

Console Label/MBean Name	Description
Unit-of-Work (UOW) Message Handling Policy `UnitOfWorkHandlingPolicy`	Specifies whether the Unit-of-Work (UOW) feature is enabled for a destination. • `Pass-Through` : By default, destinations do not treat messages as part of a UOW. • `Single Message Delivery` : Select this option if UOW consumers are receiving component messages on this terminal destination. When selected, component UOW messages are formed into a list and are consumed as an `ObjectMessage` containing the `java.util.List`.

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

Table 11-3 (Cont.) Unit-of-Work Configuration Options

Console Label/MBean Name	Description
Expiration time for incomplete UOW Messages `IncompleteWorkExpirationTime`	The maximum length of time, in milliseconds, before undeliverable messages in an incomplete UOW are expired. Such messages will then follow the expiration policy defined for undeliverable messages. Message expiration begins after the first UOW message arrives. This field is effective only if Unit-of-Work Handling Policy is set to `Single Message Delivery`. The default value, -1, disables the incomplete-work timeout. Component messages remain subject to their normal message-expiration settings. **Note:** If an expiration time is not configured on terminal destination, then it is possible for a UOW message to wait indefinitely on the destination when a component message was either: (A) never sent/committed, (B) expired, or (C) manually deleted).
Maximum messages in a Unit-of-Work `MaximumUnitOfWorkMessages`	Maximum number of component messages in a UOW at a terminal destination using Single Message Delivery. Default: 200. Legal range: 1–4096. The destination can inherit the value from its JMS template. Changes apply dynamically to subsequent components of incomplete work; an already accepted end component continues to define that work's permitted sequence range.

For more information about these parameters, see `DestinationBean` and `TemplateBean` in the *MBean Reference for Oracle WebLogic Server*.

UOW Message Routing for Terminal Distributed Destinations

The Unit-of-Order Routing field is used to determine the routing of UOW messages for uniform distributed destinations, using either the path service or hash-based routing. And similar to UOO, when a UOW terminal destination is also a distributed destination, all messages within a UOW must go to the same distributed destination member. For more information on the UOO routing mechanisms, see Using Unit-of-Order with Distributed Destinations.

However, basic UOO routing and UOW routing are not the same. Strictly, all messages within a single UOO do not have to go to the same member: when there are no more unconsumed messages for a certain UOO, newly arrived messages can go to any member. In UOW, message routing must be guaranteed until the *whole* UOW has arrived at the physical destination and consumption is irrelevant.

How to Write a UOW Consumer for a Terminal Destination

The sample in Example 11-3 receives assembled UOW messages from a terminal destination and reports the number of component messages in each list.

Example 11-3 Sample Client Code for UOW Terminal Destination

This fragment uses a configured qReceiver1 that receives assembled UOW ObjectMessages inside a method handling or declaring jakarta.jms.JMSException. The receive() call waits until a message arrives or the consumer is closed; application-specific validation of the expected component count is separate.

```
jakarta.jms.Message message;
do {
```


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

```
    message = qReceiver1.receive();
    if (message != null) {
        java.util.List<?> components = (java.util.List<?>)
            ((jakarta.jms.ObjectMessage) message).getObject();
        System.out.println("Number of component messages: " + components.size());
    }
} while (message != null);
```

Message Unit-of-Work Advanced Topics

Learn how Unit-of-Work processes messages in advanced or more complex situations.

Message Property Handling

UOW is, in effect, taking multiple messages and joining them into one. This is true whether or not the messages are delivered as one message. For example, each message will have an independent expiration time, but if one expires, none of them will ever be delivered. Therefore, as a best practice your message producers should make sure that messages that make up a UOW are as uniform as possible.

Whether component messages are delivered as parts of a single message or as many messages, it is easiest to envision them as a single *virtual* message, as well as individual messages. For example, because the messages must be seen consecutively, UOW's effect on message sorting can be viewed as determining the correct placement of the virtual message. The same is true of message selection (a consumer must see the whole group or not see the group at all); WebLogic JMS must determine whether "*consumer A* must see the virtual message" before deciding to deliver all of the messages to *consumer A*.

System-Generated Properties

Some fields of the virtual message will need to be populated independently of the component messages. For example, the virtual message cannot get its value for delivery count from a component message. This is the list of property values that are system-generated:

- Timestamp
- Delivery count (redelivered)
- Destination

Final Component Message Properties

The message properties will be derived from the component messages. However, different properties get values derived in different ways. One way to derive virtual message properties is to get their values directly from one of the component messages, (this simplifies the handling of component messages with different property values). For simplicity, the last message in the UOW is the message from which the values are derived. For example, the message priority for the virtual message will be the priority of the message marked as last (by having the property `JMS_BEA_IsUnitOfWorkEnd` set to true).

This is the list of virtual message properties that are derived from the values contained in the last message in the UOW:

- Message ID
- Correlation ID
- Priority

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

- User Properties
- User ID
- `JMSReplyTo`
- `JMSType`

Component Message Heterogeneity

Use consistent delivery-mode and expiration settings for the component messages in a UOW. Component messages remain subject to expiration; expiration of a required component can prevent delivery of the complete group. The incomplete-work timeout is a separate setting described in Configuring Terminal Destinations.

JMSReplyTo Header

The assembled UOW message copies its JMSReplyTo header from the final component message, identified by JMS_BEA_IsUnitOfWorkEnd set to true.

UOW and Uniform Distributed Destinations

As discussed in UOW Message Routing for Terminal Distributed Destinations, the Unit-of-Order Routing field is used to determine the routing mechanism for UOW messages. One other requirement for UOW in distributed destinations is that all member destinations must have the same value for the UOW Handling Policy. A configuration that is configured otherwise is invalid.

As a best practice, the use of topics (especially distributed topics) is discouraged for use as intermediate UOW destinations, because this configuration may lead to duplicate component messages.

UOW and Store-and-Forward Destinations

The WebLogic Store-and-Forward service supports UOW, with the exception that a store-and-forward (SAF) imported destination cannot be a terminal destination. However, SAF obeys the routing rules of UOW messages, just as it does for UOO messages. See Using Unit-of-Order with WebLogic Store-and-Forward.

Limitations of UOW Message Groups

Understand the limitations when using Unit-of-Work message groups.

- JMS clients created using WebLogic Server earlier than 9.0 cannot create messages that will be processed as part of a UOW.
- The JMS C JNI client is not able to process UOW messages at a terminal destination, because they are object messages. It can, however, be used as a UOW producer or on an intermediate destination.
- UOW is poorly suited for sets of large file transfers. Ideally, your messaging environment is configured for lower maximum message sizes and to facilitate the streaming transfer of large chunks of data (such as large files) from a single producer to a single consumer. UOW doesn't handle this use-case because the individual messages are accumulated into a single message on the server before they are pushed to the consumer, rather than streamed.

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

12

Using Transactions with WebLogic JMS

Learn how to use local JMS transactions, JTA user transactions, and container-managed transactions with message-driven beans.

> **Note**
>
> For the standard messaging APIs used in this chapter, see the Jakarta Messaging 3.1 specification and API documentation.

Overview of Transactions

A transaction enables an application to coordinate a group of messages for production and consumption, treating messages sent or received as an atomic unit.

When a transaction commits successfully, its received messages are acknowledged and its sent messages become eligible for delivery, subject to their delivery and expiration settings. Rolling back discards its sends and makes its received messages eligible for redelivery, subject to the destination's redelivery and expiration policies.

A rolled-back queue message may be redelivered to another eligible consumer on that queue. A rolled-back topic message remains associated with its subscription; for a shared subscription, another consumer on that subscription may receive it.

For example, when shopping online, you select items and store them in an online shopping cart. Each ordered item is stored as part of the transaction, but your credit card is not charged until you confirm the order by checking out. At any time, you can cancel your order and empty your cart, rolling back all orders within the current transaction.

This chapter describes these transaction approaches:

- Use a local JMS transaction to group sends and receives within one session.
- Use an application-managed JTA transaction, where permitted, to coordinate JMS work with other participating resources.
- Use a message-driven bean with container-managed transactions for transactional asynchronous message processing.

The following sections describe these approaches.

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

> **Note**
>
> For standalone clients, a connection ExceptionListener can report asynchronous connection problems. Handle exceptions from messaging and transaction-completion calls as well; the listener does not replace that error handling. See Defining a Connection Exception Listener. EJB and servlet applications must follow their container's restrictions on listener registration.
>
> In a locally transacted session, Message.acknowledge() has no effect, and Session.recover() throws jakarta.jms.IllegalStateException. Complete the transaction with Session.commit() or Session.rollback(). For JMS work enlisted in JTA, completion is controlled by the JTA transaction.

Using JMS Transacted Sessions

A locally transacted JMS session groups its sends and receives in a transaction controlled by that session.

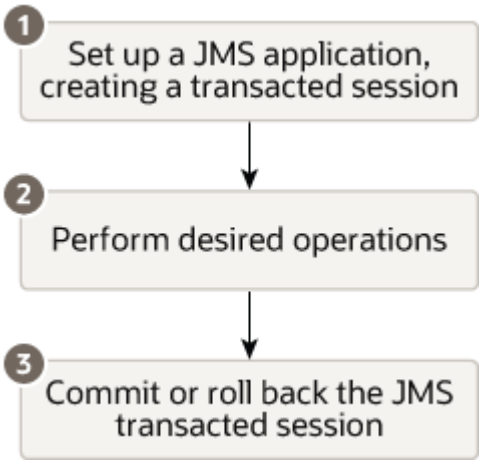
The following procedure describes an application-managed local JMS transaction. Its commit or rollback affects the sends and receives on that session, not independent database work. A native WebLogic JMS locally transacted session uses its own transaction rather than the calling thread's JTA transaction. For sessions used through Jakarta EE resource references in EJBs or servlets, follow the container transaction rules described in Automatically Enlisting Transactions.

Local JMS transactions begin implicitly. After a local transaction is committed or rolled back, the session begins the next local transaction automatically.

Before using a locally transacted session, review the connection factory's Transaction Timeout setting and adjust it if necessary for the application.

Figure 12-1 shows the steps required to set up and use a JMS transacted session.

Figure 12-1 Setting Up and Using a JMS Transacted Session



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

Step 1: Set Up JMS Application, Creating Transacted Session

Set up the JMS application as described in Setting Up a JMS Application, when creating sessions, as described in Step 3: Create a Session Using the Connection, specify that the session is to be transacted by setting the `transacted` Boolean value to `true`.

For example, the following methods show how to create a transacted session for the point-to-point and Publish/subscribe messaging models, respectively:

```
qsession = qcon.createQueueSession(
 true,
 Session.AUTO_ACKNOWLEDGE
);

tsession = tcon.createTopicSession(
 true,
 Session.AUTO_ACKNOWLEDGE
);
```

After a session is defined, you can determine whether or not a session is transacted using the following session method:

```
public boolean getTransacted(
) throws JMSException
```

> **Note**
>
> The acknowledge value is ignored for transacted sessions.

Step 2: Perform Desired Operations

Perform the desired operations associated with the current transaction.

Step 3: Commit or Roll Back the JMS Transacted Session

After you have performed the desired operations, execute one of the following methods to commit or roll back the transaction.

To commit the transaction, execute the following method:

```
public void commit(
) throws JMSException
```

The `commit()` method commits all messages sent or received during the current transaction. Sent messages are made visible, while received messages are removed from the messaging system.

To roll back the transaction, execute the following method:

```
public void rollback(
) throws JMSException
```

The `rollback()` method cancels any messages sent during the current transaction and returns any messages received to the messaging system.

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

If either the `commit()` or `rollback()` methods are issued outside of a JMS transacted session, then a `jakarta.jms.IllegalStateException` is thrown.

Using JTA User Transactions

Jakarta Transactions (JTA) coordinates transactions across participating resources. Applications can use UserTransaction where programmatic transaction demarcation is permitted, or use container-managed transactions.

Use jakarta.transaction.UserTransaction to begin, commit, and roll back an application-managed JTA transaction. See the Jakarta Transactions 2.0 UserTransaction API. In an EJB, use UserTransaction only with bean-managed transaction demarcation; container-managed beans rely on their configured transaction attributes. See Transactions in EJB Applications in *Developing JTA Applications for Oracle WebLogic Server*.

A native WebLogic JMS locally transacted session does not join a calling thread's JTA transaction. For the direct JMS client procedure below, create a nontransacted session and use a WebLogic JMS connection factory with XA Connection Factory Enabled set to true.

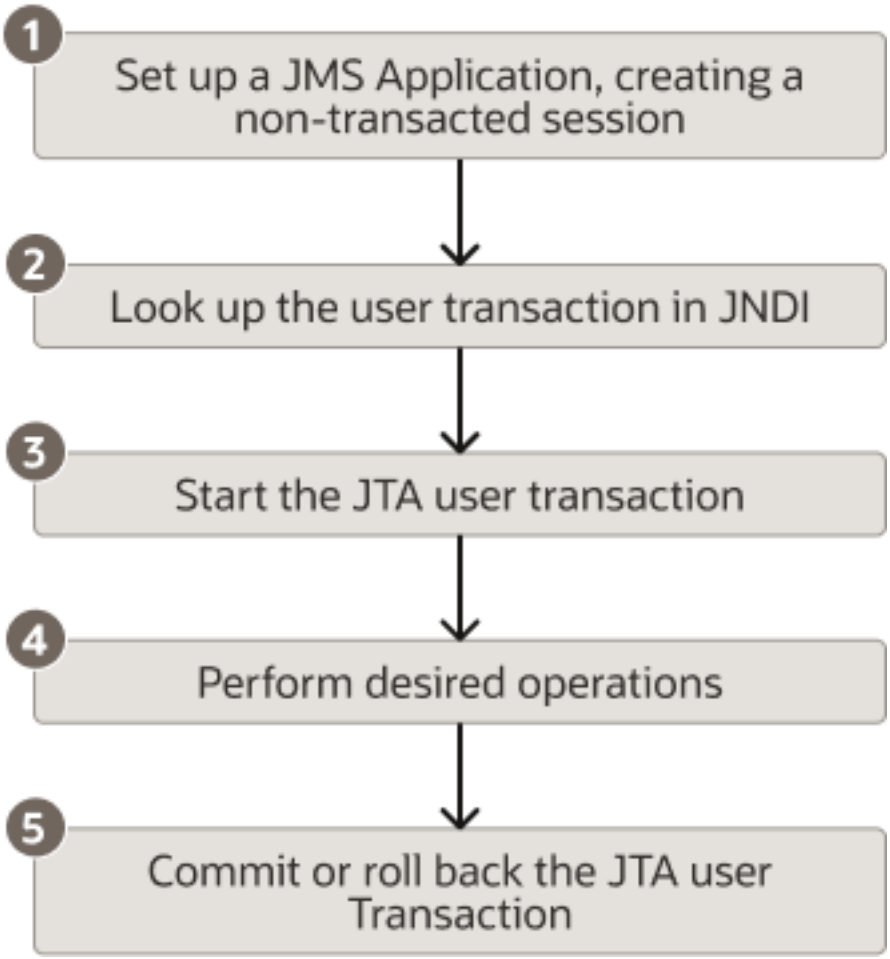
WebLogic Server supports two-phase commit to coordinate a JTA transaction across participating resource managers. Applications must handle transaction-completion exceptions: a commit request can result in rollback, and a heuristic outcome can require administrative resolution.

The XA Connection Factory Enabled setting allows nontransacted sessions created by that factory to participate in the calling thread's JTA transaction. Begin the transaction before the JMS operations that must participate. For container-managed resource references, also follow Automatically Enlisting Transactions.

Figure 12-2 shows the steps required to set up and use a JTA user transaction.

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

Figure 12-2 Setting Up and Using a JTA User Transaction



Step 1: Set Up JMS Application, Creating Non-Transacted Session

Set up the JMS application as described in Setting Up a JMS Application, however, when creating sessions, as described in Step 3: Create a Session Using the Connection, specify that the session is to be non-transacted by setting the `transacted` boolean value to `false`.

For example, the following methods illustrate how to create a non-transacted session for the PTP and Pub/sub messaging models, respectively.

```
qsession = qcon.createQueueSession(
 false,
 Session.AUTO_ACKNOWLEDGE
);

tsession = tcon.createTopicSession(
 false,
 Session.AUTO_ACKNOWLEDGE
);
```


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

> **Note**
>
> When a session's JMS operations participate in an active JTA transaction, transaction completion controls their acknowledgement; the session's acknowledgement mode does not control those operations.

Step 2: Look Up the User Transaction in JNDI

The application uses JNDI to return an object reference to the `UserTransaction` object for the WebLogic Server domain.

For a WebLogic client using an established JNDI Context named ctx, look up the UserTransaction as follows. The enclosing method must handle or declare javax.naming.NamingException.

```
jakarta.transaction.UserTransaction xact =
    (jakarta.transaction.UserTransaction)
        ctx.lookup("jakarta.transaction.UserTransaction");
```

In a Jakarta EE component that permits UserTransaction access, use injection or the component-environment name java:comp/UserTransaction.

Step 3: Start the JTA User Transaction

Start the JTA user transaction using the `UserTransaction.begin()` method. For example:

```
xact.begin();
```

Step 4: Perform Desired Operations

Perform the desired operations associated with the current transaction.

Step 5: Commit or Roll Back the JTA User Transaction

Once you have performed the desired operations, execute one of the following `commit()` or `rollback()` methods on the `UserTransaction` object to commit or roll back the JTA user transaction.

To commit the transaction, execute the following `commit()` method:

```
  xact.commit();
```

The `commit()` method asks the transaction manager to complete the current transaction across its enlisted resources. Handle the declared completion exceptions; calling commit() does not by itself guarantee that the transaction committed.

To roll back the transaction, execute the following `rollback()` method:

```
  xact.rollback();
```

The `rollback()` method causes WebLogic Server to call the Transaction Manager to cancel the transaction, and roll back all operations performed during the current transactions.

Once you call the `commit()` or `rollback()` method, you can optionally start another transaction by calling `xact.begin()`.

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

JTA Transactions Using Message-Driven Beans

Use a message-driven bean with container-managed transactions to process asynchronously delivered messages transactionally.

For a JMS message-driven bean using container-managed transactions and the Required transaction attribute, the container establishes the transaction for the message-listener invocation. Message receipt and enlisted resource operations participate in that transaction. The bean does not call UserTransaction.begin() to establish this container-managed transaction.

A bean-managed transaction started inside onMessage() does not include receipt of the triggering message. Use container-managed transactions when message receipt and processing must share the transaction.

See Designing Message-Driven EJBs in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Example: JMS and EJB in a JTA User Transaction

Learn how to set up an application for mixed EJB and JMS operations in a JTA user transaction.

These fragments illustrate a standalone WebLogic client that uses an application-managed JTA transaction for JMS operations and calls to an EJB configured to participate in that transaction. Use a JMS connection factory with XA Connection Factory Enabled set to true. The client must establish its JNDI context, connection, session, and EJB reference, start the connection before receiving messages, and handle naming, messaging, and transaction exceptions. These fragments omit application-specific operations and error handling; they are not a complete executable application.

> **Note**
>
> When the WebLogic Server examples are installed, see the JMS/JDBC transaction example in WL_HOME/samples/server/examples/src/examples/jta/jmsjdbc, where WL_HOME is the WebLogic Server installation directory.

Import the required Java and Jakarta APIs, including the `jakarta.transaction.UserTransaction` interface.

```
import java.io.*;
import java.util.*;
import jakarta.transaction.UserTransaction;
import javax.naming.*;
import jakarta.jms.*;
```

Define the WebLogic JNDI lookup-name constant.

```
public static final String JTA_USER_XACT =
    "jakarta.transaction.UserTransaction";
```


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

Step 1 Set Up the JMS Application

Set up the JMS application, creating a non-transacted session. For more information on setting up the JMS application, refer to Setting Up a JMS Application.

```
//JMS application setup steps including, for example:
 qsession = qcon.createQueueSession(false,
                Session.CLIENT_ACKNOWLEDGE);
```

Step 2 Look Up the User Transaction

Look up the `UserTransaction` using JNDI.

```
UserTransaction xact = (UserTransaction)
        ctx.lookup(JTA_USER_XACT);
```

Step 3 Start the JTA User Transaction

Start the JTA user transaction.

```
xact.begin();
```

Step 4 Perform the Desired Operations

Perform the desired operations.

```
// Perform some JMS and EJB operations here.
```

Step 5 Commit the JTA User Transaction

Commit the JTA user transaction.

```
xact.commit();
```

Using Cross-Domain Security

For transactions involving multiple domains, configure the participating domains for compatible, secure transaction communication.

Review Cross Domain Security together with the effective JTA Security Interoperability Mode for the participating servers, including applicable cluster-level JTA configuration. These settings have different purposes; do not treat them as interchangeable enable flags. See Configuring Secure Inter-Domain and Intra-Domain Transaction Communication in *Developing JTA Applications for Oracle WebLogic Server*.

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

13

Developing Advanced Pub/Sub Applications

Understand the advanced WebLogic JMS publish and subscribe (pub/sub) concepts and functionality of Uniform Distributed Topics (UDTs) necessary to design high availability (HA) applications.

Overview of Advanced High Availability Concepts

WebLogic messaging offers high availability and scalability by using distributed destinations. The WebLogic Server migration features also provide high availability for the individual members of a distributed destination.

> **Note**
>
> Oracle recommends designing applications that utilize WebLogic Server MDBs or the Oracle SOA JMS Adapter rather than explicitly handling all potential topology changes.

WebLogic Messaging High Availability Features

Oracle's WebLogic messaging offers high availability (HA) and scalability using the following features:

- Using Distributed Destinations
- Migration of JMS-related Services in *Administering JMS Resources for Oracle WebLogic Server*
- Whole Server Migration in *Administering Clusters for Oracle WebLogic Server*

Distributed Destinations make a group of JMS physical destinations accessible as a single, logical destination to a client. Applications that use distributed destinations usually have higher availability and better scalability because WebLogic JMS provides load balancing and failover among member destinations of a distributed destination within a cluster. Automatic Service Migration (ASM) and Whole Server Migration (WSM) enable restarting either a set of services (including JMS servers and destinations) or an entire WebLogic Server instance in a new location. These migration features provide high availability for the individual members of a distributed destination.

The nature of these technologies means that the topology of a JMS system can be unknown to a client application as:

- The scaling of a cluster, along with the scaling of a distributed destination may exceed the number of consumers defined by the application.
- The topology may dynamically change in the event of a server or service failure.

Typically, topology changes are handled transparently using MDBs either locally or on a remote WebLogic Server instance. However, when using other client types, these topology changes

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

must be explicitly handled by the application, especially if the application is remote to the servers hosting the JMS destinations.

Application Design Limitations When Using Replicated Distributed Topics

Applications implementing Uniform Distributed Topics earlier than WebLogic Server 10.3.4.0 were constrained by the following limitations:

- Messages are always forwarded and duplicated across a distributed topic, which means that either parallel processing, and/or ensuring that a clustered application gets one copy of each message, may requires significant additional configuration, coding, and message hops.
- Only one consumer at a time can process the messages in a given subscription except for the limited case of Non-XA MDBs where all processing of the subscription must occur on the same server with a thread pool. This prevents most customers from designing application architectures that intend to have "round-robin" distributed or parallel processing of a single subscription's topic messages, instead of single-threaded processing.
- MDBs only directly support durable subscriptions on distributed topics that are located in the same cluster.
- For applications other than MDBs, a durable subscriber created for a distributed topic can only be created on a distributed topic (DT) member, and the durable subscription will only exist on that member. If the member hosting the subscription is down, then the subscription will not be available to any subscriber (and is therefore not "highly available" by definition).
- Pinning subscribers to a distributed topic member prevents automatic adjustment to changes in topology in the same way that adjustments are made for distributed queues.

Advanced Topic Features

Starting in WebLogic Server 10.3.4.0, partitioned distributed topics, combined with the ability to share subscriptions and allow multiple connections to use the same Client ID, provide the following application design patterns that provide parallel processing and HA capabilities similar to distributed queues:

- One copy per instance: Each instance of an application gets one copy of each message that is published to the Topic.
- One copy per application: Each application as a whole (that is all instances of the application together) receives one copy of each message that is published to the Topic. Each instance only receives a subset of the messages that are sent to the Topic.

> **Note**
>
> Oracle recommends designing applications that utilize WebLogic Server MDBs. See Configuring and Deploying MDBs Using JMS Topics in *Programming Message-Driven Beans for Oracle WebLogic Server* for detailed information about how to design and implement applications that use message-driven beans to provide improved HA and scalability.

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

Advanced Topic Messaging Features for High Availability

In order to understand how an application can achieve One-copy-per-instance and One-copy-per-application design patterns, you need to understand the new and changed features such as shared subscriptions and the client ID policy.

Shared Subscriptions and Client ID Policy

Before WebLogic Server 10.3.4.0, one subscription, durable or non durable, could only be accessed by a single subscriber instance at any time. Each subscriber receives all messages that are sent to a topic after the subscription is established and the messages for each subscription are processed sequentially by one consumer.

In this WebLogic Server release, multiple subscribers can share one subscription (durable or non durable). Messages are distributed among multiple consumers that share the same subscription and can be processed in parallel. Subscription sharing only occurs on the same destination instance or the same member instance of a DT. See Configure Shared Subscriptions in *Administering JMS Resources for Oracle WebLogic Server*.

WebLogic supports policy-based subscription sharing and the standard named shared-subscription APIs. For policy-based sharing through the traditional subscriber APIs, configure a client ID and the Sharable Subscription Sharing Policy. The standard Session.createSharedConsumer() and Session.createSharedDurableConsumer() APIs instead take a subscription name; they do not require a client ID or a Sharable connection-factory setting. Distributed-topic restrictions are described in Considerations When Using Standard Shared Subscriptions.

Client ID Policy controls reuse of a client ID by connections. Restricted, the default, prevents another Restricted connection in the cluster from using that client ID concurrently. Unrestricted permits reuse. The policies also distinguish subscriptions; do not treat otherwise identical Restricted and Unrestricted subscriptions as the same subscription. See How Sharing a Durable Subscription Works.

What Is the Subscription Key

A subscription's identity depends on the API and client ID policy used to create it. For WebLogic policy-based non-durable sharing, consumers must use the same client ID and client ID policy, and must match the physical topic, message selector, and noLocal setting. A different physical topic, selector, or noLocal setting creates a separate non-durable subscription.

For durable subscriptions, the subscription name and client ID, when present, identify the subscription within the applicable client-ID-policy scope. With Unrestricted client ID policy, matching names and client IDs can identify separate subscriptions on different physical topics or distributed-topic members. Restricted subscriptions do not acquire that member-local identity simply by reusing the same name.

Standard named shared subscriptions use a subscription name and, when configured, a client ID. Follow the matching rules of the standard creation API and the WebLogic distributed-topic restrictions described in Considerations When Using Standard Shared Subscriptions.

Configuring a Shared Subscription

For WebLogic policy-based sharing, set Subscription Sharing Policy to Sharable on the connection factory, or use the applicable WebLogic connection-level override before creating

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

consumers. Consumers from different connection factories can share a subscription when their effective policies and subscription details match. Sharing is local to a physical topic or distributed-topic member. See What Is the Subscription Key and `SubscriptionSharingPolicy` in *MBean Reference for Oracle WebLogic Server*.

How Sharing a Non-Durable Subscription Works

This section describes WebLogic policy-based non-durable sharing through the traditional subscriber APIs. Set a client ID and the Sharable Subscription Sharing Policy. Without a client ID, those APIs create separate non-durable subscriptions even when the policy is Sharable. Standard named shared subscriptions can omit the client ID; see Considerations When Using Standard Shared Subscriptions.

The first subscriber that is created with a subscription key creates the subscription. All subsequently created subscribers with the same subscription key share the subscription created by the first subscriber if all subscription details (such as: the selector, `noLocal` option, and the physical destination) match. For example:

- If a subscription is created with a selector and `noLocal` option, a subscriber creation call that uses the same subscription key but a different selector, `noLocal` option or a different physical destination is treated as a different subscription.
- Multiple Exclusive non-durable consumers with otherwise matching subscription details receive through independent subscriptions. Mixing Exclusive and Sharable consumers with matching details can instead fail because of a policy conflict; see How a Shared Subscription Policy for a Non-Durable Subscription Is Determined.

> **Note**
>
> Connections using the same Restricted client ID cannot coexist. Use the same connection, or use Unrestricted client ID policy consistently when separate connections need the same client ID.

How a Shared Subscription Policy for a Non-Durable Subscription Is Determined

For WebLogic policy-based non-durable sharing, matching Sharable consumers join the same subscription on the physical topic. While matching consumers are active, creating a consumer with a conflicting Exclusive or Sharable policy fails with a jakarta.jms.JMSException.

Multiple Exclusive non-durable consumers are permitted with otherwise matching subscription details; each uses an independent subscription. An Exclusive consumer therefore does not join the existing consumer's subscription. This differs from an exclusive durable subscription, which cannot have multiple active consumers.

Closing the last consumer removes a non-durable subscription. Creating consumers afterward creates new subscriptions according to their effective settings.

How a Non-Durable Subscription Is Closed

When the last consumer of a non-durable subscription closes, the subscription and its associated resources are removed.

There is no runtime MBean that represents a non-durable subscription, regardless of whether it is a shared or exclusive subscription. It is possible to monitor individual subscribers using the appropriate `JMSConsumerRuntime` MBean.

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

How Sharing a Durable Subscription Works

This section describes WebLogic policy-based durable sharing. Consumers share an existing durable subscription only when its identity and subscription details match and the effective sharing policy permits sharing. A Restricted client ID and subscription name identify the existing subscription within the cluster. With Unrestricted client ID policy, the same client ID and subscription name can identify separate subscriptions on different physical topics or distributed-topic members; consumers share only the subscription on the same physical destination.

Matching subscription details include the selector and noLocal setting. Changing either setting is a subscription change, not a way to attach another consumer to the unchanged subscription.

A creation request that changes subscription details fails with a jakarta.jms.JMSException while the existing subscription is in use. When the old subscription can be deleted, replacing it discards its retained messages. Close its consumers and complete outstanding transactions before attempting such a change; absence of active consumers alone does not guarantee that deletion can proceed.

Changing only Subscription Sharing Policy after all consumers close does not delete the subscription's retained messages. See How a Shared Subscription Policy for a Durable Subscription Is Determined. For standard named shared subscriptions, see Considerations When Using Standard Shared Subscriptions.

How a Shared Subscription Policy for a Durable Subscription Is Determined

For WebLogic policy-based durable sharing, the first active consumer establishes the subscription's sharing policy. While the subscription has active consumers, a request with a conflicting policy fails with a jakarta.jms.JMSException. After all consumers close, a new consumer can establish a different sharing policy.

An Exclusive durable subscription permits only one active consumer. A Sharable durable subscription permits multiple matching Sharable consumers; it does not permit an Exclusive consumer to join while those consumers remain active.

> **Note**
>
> Changing the Subscription Sharing Policy on an existing durable subscription does not delete any messages that already exist on the subscription.

How to Unsubscribe a Durable Subscription

Before unsubscribing a subscription, you must consider the Client ID Policy for the subscription:

- Applications that use a client ID Policy with a value of `RESTRICTED` unsubscribe a durable subscription using the standard `Session.unsubscribe(String name)` API.

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

> **Note**
>
> Before WebLogic Server 10.3.4.0, all client IDs are `RESTRICTED` by default. A client ID could only be used by one connection at any given time in a WLS JMS cluster.

- For Unrestricted client ID policy, use `WLSession.unsubscribe(Topic topic, String name)`. Supply the physical topic or distributed-topic member first, followed by the subscription name. A logical distributed-topic object or a null topic is not valid for this call.

Considerations When Unsubscribing a Durable Subscriber

The following section provides information on how to unsubscribe or avoid scenarios that throw an exception:

- If there are active consumers on the subscription, a call to the `unsubscribe()` method throws a `JMSException`.
- Close all consumers and complete outstanding transactions before unsubscribing. Uncommitted or unacknowledged receives can prevent deletion even after the consumers have closed. A successful unsubscribe removes the matching durable subscription and its retained messages; handle JMSException if the operation cannot complete.
- The `unsubscribe()` method of a durable subscription is done per standalone topic or per member of a DT.
- A subscription created using a connection with a `RESTRICTED` client ID can only be cleaned up from a connection that uses the same `RESTRICTED` Client ID.
- A subscription created using a connection with an `UNRESTRICTED` client ID can only be cleaned up from a connection using the same `UNRESTRICTED` client ID.
- If WebLogic JMS does not find a matching subscription on the topic that was created with the same client ID and client ID Policy as the `unsubscribe` call, then an `InvalidDestinationException` is thrown.
- If an `unsubscribe` call with an `UNRESTRICTED` client ID specifies a DT or does not specify any Topic, then an `InvalidDestinationException` is thrown.
- Although .Net and C API messaging applications can share subscriptions by using the client ID Policy and Subscription Sharing Policy on a connection factory deployed on WebLogic Server 10.3.4.0 or later, an unsubscribe API extension is not yet available for subscriptions that use an unrestricted client ID. The workaround is to use administrative measures described in Managing Durable Subscriptions.

Managing Durable Subscriptions

A durable subscription that is no longer needed can continue to accumulate messages after its consumers stop. Such subscriptions can consume quota and storage and may eventually prevent further sends. Confirm that a subscription is no longer required before deleting it and its retained messages.

For example, the `unsubscribe` call fails when there are active subscribers on the subscription and the `unsubscribe` call does not reach subscriptions on inactive (shutdown) members. When this happens, the subscription is left on the member where the call failed until it is manually removed by an administrator or the call is repeated.

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

To help handle these situations, administrators have the following options to monitor and manage durable subscriptions:

- There is one instance of the `JMSDurableSubscriberRuntimeMBean` for each durable subscription. Administrators can monitor a topic or UDT using the WebLogic Remote Console or by using WLST command line or scripts. See Monitor Servers in *Oracle WebLogic Remote Console Online Help*.
- Use LastMessagesReceivedTime together with consumer activity and message counts to investigate a potentially abandoned subscription. LastMessagesReceivedTime reports the last successful delivery to a subscriber; if no message has been delivered since the subscription was created or recovered, it reports that creation or recovery time. MessagesCurrentCount describes messages still available, while MessagesPendingCount describes uncommitted or unacknowledged messages. An old timestamp or an inactive consumer alone does not prove that a durable subscription is abandoned. Confirm application ownership and the disposition of retained messages before removing the subscription.

Naming Conventions for JMSDurableSubscriberRuntimeMBean

If a durable subscription is created using the subscription key, `<MyClientID, MySubscriptionName>`, then the name of the associated `JMSDurableSubscriberRuntimeMBean` is either:

- *`MyClientID_MySubscriptionName`* when the client ID Policy is `RESTRICTED`. Where *`MyClientID`* is the Client ID for this subscription, and *`MySubscriptionName`* is the name of the subscription.
- *`MyClientID_MySubscriptionName@topicName@JMSServerName`* when the client ID Policy is `UNRESTRICTED`. Where *`MyClientID`* is the client ID for this subscription, *`MySubscriptionName`* is the name of the subscription., *`topicName`* is the name of a standalone topic or a member of a UDT, and *`JMSServerName`* is the name of the JMS Server that the topic or member is deployed on.

Design Strategies When Using Topics

Learn about the Topic-based design strategies that can be used to develop high availability applications.

One-Copy-Per-Instance Design Strategy

With the one-copy-per-instance design strategy, each application instance has its own subscription and receives its own copy of messages published to the topic, subject to normal JMS delivery and message-selection rules. This strategy is appropriate when every application instance needs to process messages independently.

This pattern is usually best implemented using an MDB configured for the intended strategy. The MDB container sets up the required policies and subscriptions across the cluster. See Best Practices for Distributed Topics.

For applications that do not use MDBs, see Shared Subscriptions and Client ID Policy and Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API for information about subscription sharing and distributed-topic member availability.

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

One-Copy-Per-Application Design Strategy

With the one-copy-per-application design strategy, application instances share subscriptions so that each message published to the topic is delivered to the application as a whole rather than separately to every instance, subject to normal JMS delivery and message-selection rules. Messages can be distributed among the instances, allowing them to share the work of processing messages.

This pattern is usually best implemented using an MDB configured for the intended strategy. The MDB container sets up the required policies and subscriptions across the cluster. See Best Practices for Distributed Topics.

For applications that do not use MDBs, see Shared Subscriptions and Client ID Policy and Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API for information about subscription sharing and distributed-topic member availability.

Considerations When Using Standard Shared Subscriptions

The named shared-subscription APIs introduced in JMS 2.0 remain part of Jakarta Messaging. The following WebLogic restrictions apply to distributed topics.

Observe these rules when using the standard shared-subscription APIs:

- For a shared non-durable subscription on a logical distributed topic or an individual distributed-topic member, use Unrestricted Client ID Policy when the connection has no client ID.
- Do not pass a logical distributed-topic object directly to Session.createSharedDurableConsumer(). Create durable subscriptions on individual members, or use MDBs or the documented WebLogic helpers to manage member subscriptions.
- For a shared durable subscription on an individual distributed-topic member, use Unrestricted Client ID Policy when the connection has no client ID.

> **Note**
>
> For Unrestricted client ID policy, use `WLSession.unsubscribe(Topic topic, String name)`. Supply the physical topic or distributed-topic member first, followed by the subscription name. A logical distributed-topic object or a null topic is not valid for this call.

Replacing a Replicated Distributed Topic

Learn about replacing a Replicated Distributed Topic (RDT) with a standalone topic or PDT.

Reasons for Replacing a Replicated Distributed Topic

It is sometimes necessary to replace an existing Replicated Distributed Topic (RDT) with a Partitioned Distributed Topic (PDT) or standalone topic because RDTs are not supported in the following scopes:

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

- Cluster-targeted JMS servers. See Simplified JMS Cluster and High Availability Configuration in *Administering JMS Resources for Oracle WebLogic Server*.
- Dynamic clusters

An attempt to configure or deploy an RDT in any of these scopes generates a configuration validation error.

It also can be helpful to replace an RDT with a PDT or standalone topic because these options handle the same use cases, yet are simpler and tend to perform better. RDTs implicitly run transactional internal forwarders to duplicate messages between their members, and these forwarders have a relatively high overhead.

Important Prerequisites Before Replacing an RDT

Plan the replacement so that required messages on the existing replicated distributed topic are processed before it is retired. Coordinate producers and consumers, complete outstanding transactions, and identify every durable subscription that must be retained or recreated.

For a side-by-side replacement, create the new topic with a different name and configure its required subscriptions and consumers before directing production traffic to it. Coordinate the cutover to avoid unintended loss or duplicate processing. After required messages on the old topic have been processed, close its consumers, remove only the obsolete durable subscriptions using the supported unsubscribe or administrative operations, and retire the old topic configuration.

Do not delete persistent-store files or database tables as a shortcut for removing old topic subscriptions. A store can contain other required resources. See Managing Durable Subscriptions.

Replacing an RDT with a Standalone Topic

Replacing an RDT with a singleton standalone topic instead of a PDT can be the simplest option, but sacrifices scalability and some HA. Ensuring that a cluster-hosted standalone topic is migratable, can mitigate HA concerns. See What About Failover? in *Administering JMS Resources for Oracle WebLogic Server*.

> **Note**
>
> Standalone topics that are hosted on cluster-targeted JMS servers or a dynamic cluster can only be hosted on a JMS server that references a store configured with a singleton distribution policy. They also require configuring cluster leasing on a cluster where database leasing is recommended over consensus leasing. See Simplified JMS Cluster and High Availability Configuration in *Administering JMS Resources for Oracle WebLogic Server*.

Replacing an RDT with a PDT

To configure a PDT, set the `forwarding-policy` attribute of a uniform distributed topic to `Partitioned` instead of `Replicated`. A PDT does not duplicate a message produced to one of its members to every other member, so this different semantic may require further changes:

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

- For MDB consumers of a PDT, configure the activation-config property `topicMessagesDistributionMode` as `One-Copy-Per-Server` or `One-Copy-Per-Application`, according to the intended delivery pattern. Compatibility mode is not supported for PDTs. See Configuring and Deploying MDBs Using JMS Topics and Topic Deployment Scenarios in *Developing Message-Driven Beans for Oracle WebLogic Server*.
- If you are using the SOA JMS adapter to consume from a PDT, then no change is needed. It defaults to the MDB equivalent of `one-copy-per-app` when consuming from a PDT. See Accessing Distributed Destinations (Queues and Topics) on the WebLogic Server JMS in *Understanding Technology Adapters*.
- For OSA JMS Adapter consumers, check the adapter release's supported PDT configuration and ensure that its subscriptions cover the required PDT members before replacing the topic.
- If you have `jakarta.jms` topic consumers that are not an MDB, SOA JMS Adapter, or OSA JMS Adapter, then application code changes may be needed. For example, the application may need to be changed so that it consumes from subscriptions on each and every PDT member instead of from a single subscription. This is because a PDT does not replicate each sent message to each of its members. For more information and a discussion of helper APIs in this area, see Developing Advanced Pub/Sub Applications, Advanced Programming with Distributed Destinations and Using the JMS Destination Availability Helper API.

Best Practices for Distributed Topics

Follow Oracle's recommendations when designing new applications using distributed topics.

- Simplify application design and complexity by utilizing MDBs. See:
  - Distributed Topic Deployment Scenarios in *Developing Message-Driven Beans for Oracle WebLogic Server*
  - Configuring and Deploying MDBs Using Distributed Topics in *Developing Message-Driven Beans for Oracle WebLogic Server*
- If MDBs are not an option, consider using an `UNRESTRICTED` Client ID Policy, a `SHARABLE` Subscription Policy, in combination with a Partitioned Topic (a distributed topic with a `PARTITIONED` forwarding policy). See:
  - Configure an Unrestricted ClientID in *Administering JMS Resources for Oracle WebLogic Server*
  - Configure Shared Subscriptions in *Administering JMS Resources for Oracle WebLogic Server*
  - Configuring Partitioned Distributed Topics in *Administering JMS Resources for Oracle WebLogic Server*
  - Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API

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

14

Recovering from a Server Failure

Understand how WebLogic JMS client applications reconnect or recover from a server/network failure and learn how to migrate JMS data after a server failure.

Automatic JMS Client Failover

WebLogic JMS automatic client reconnection is deprecated and disabled by default. Applications should implement explicit recovery from connection failures.

> **Note**
>
> The deprecated reconnect settings do not provide automatic reconnection in the default configuration of this release. Applications must handle messaging exceptions and recover their connections, sessions, producers, and consumers as appropriate. See Client Resiliency Best Practices in *Administering JMS Resources for Oracle WebLogic Server*.

The following automatic-reconnect subsections describe legacy behavior only when the deprecated reconnect mechanism is active. They do not describe the default recovery behavior of this release. Setting ReconnectPolicy on a connection factory or calling setReconnectPolicy() is not, by itself, sufficient to activate that mechanism in the default configuration.

A client connection failure and migration of a JMS server are separate events. Applications must handle failures even when the destination service is configured for high availability. For new application recovery logic, follow the linked client-resiliency guidance.

Automatic Reconnect Limitations

A connection failure can leave the application uncertain whether a send completed. Handle the exception and determine whether retrying is safe under the application's transaction and duplicate-handling design. Retrying can produce another copy of a message.

For retryable business operations, use a stable application identifier and duplicate-aware or idempotent processing. Do not use a queue browse or the absence of a message as proof that the send failed; the message may already have been consumed or expired.

> **Note**
>
> An unavailable destination can delay investigation or recovery. Its return to service does not, by itself, resolve whether an earlier send completed.

The deprecated reconnect mechanism has additional limitations:

- Queue browsers: `jakarta.jms.QueueBrowser`

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

- The legacy IIOP JMS thin client, wljmsclient.jar, does not participate in this automatic reconnect mechanism. Do not confuse it with the T3 thin client, wlthint3client.jar.
- Client statistics are reset on each reconnect, which results in the loss of historical data for the client.
- Under some circumstances, automatic reconnect is not possible. If it is not possible, an exception is reported.
- Temporary destinations (`jakarta.jms.TemporaryQueue` and `jakarta.jms.TemporaryTopic`).

> **Note**
>
> Do not rely on a temporary destination surviving connection recovery. Its lifetime is associated with its creating connection, and a server or connection failure can invalidate it. Recreate temporary destinations and update any request/reply references as required by the application's recovery design.

Automatic Failover for JMS Producers

When the deprecated automatic-reconnect mechanism is active and its effective policy includes producers, eligible Connection, Session, and MessageProducer objects can be refreshed after a connection failure. Recovery is not guaranteed for every failure, and operations can still throw exceptions. Automatic producer reconnection is disabled by default in this release. See Automatic JMS Client Failover.

Sample Producer Code

Example 14-1 is pseudocode illustrating producer resource use and failure handling. It is not executable Java and does not implement a complete recovery loop. The deprecated automatic-reconnect mechanism is disabled by default.

Example 14-1 Sample JMS Client Code for Message Production

```
1. Create a JNDI context using the application's connection settings.
2. Look up the connection factory and destination.
3. Create a connection, a nontransacted session, and a producer.
4. Register an exception listener that signals the application's recovery coordinator.
5. For each message:
   a. Create and send the message.
   b. If a JMS operation fails, stop using the failed resources and invoke
      the application's recovery and uncertain-send handling.
      Do not blindly resend a message whose outcome is unknown.
6. Close the connection and JNDI context on completion or failure.
```

Application recovery logic must coordinate resource cleanup, reconnection attempts, and safe handling of an uncertain send outcome. See Automatic Reconnect Limitations and Client Resiliency Best Practices in Administering JMS Resources for Oracle WebLogic Server.

Reusable ConnectionFactory Objects

A `ConnectionFactory` object looked up using JNDI is reusable after a server or network failure without requiring a re-lookup. A network failure could be between the JMS client JVM and the remote WebLogic Server instance it is connected to as part of the JNDI lookup, or between the JMS client JVM and any remote WebLogic Server instance in the same cluster where the JMS

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

client subsequently connects. See Example 14-1 and Example 14-2. Reusing the object does not guarantee that the corresponding service is currently available; handle failures from subsequent operations.

Reusable Destination Objects

A destination object (queue or topic) looked up using JNDI is reusable after a server or network failure without requiring another lookup. The same principle applies to producers that send to a distributed destination, because the client looks up the distributed destination in JNDI, and not the unavailable distributed member. See Example 14-1 and Example 14-2. Reusing the object does not guarantee that the corresponding service is currently available; handle failures from subsequent operations.

A network failure could be between the client JVM and the WebLogic Server instance it is connected to, or between that WebLogic Server instance and the WebLogic Server instance that actually hosts the destination. The Destination object will also be robust after restarting the WebLogic Server instance hosting the destination.

> **Note**
>
> For information on how consumers of distributed destinations behave with automatic JMS client reconnect, see Consumers of Distributed Destinations.

Reconnected Connection Objects

A JMS Connection is a logical messaging connection; it does not necessarily correspond to a dedicated physical network connection. When the deprecated reconnect mechanism is active and refresh succeeds, the same application-visible Connection object can communicate through a replacement server-side connection.

Eligible derived objects are refreshed according to the effective reconnect policy and the limitations described in this section. A synchronous operation can wait for reconnect processing according to `ReconnectBlockingMillis`, and can still fail. This setting is not a general timeout for every messaging or network operation.

Background retry duration is governed by TotalReconnectPeriodMillis. Its legacy default of -1 permits indefinite retries; a finite setting limits the retry period. Closing the connection stops its reconnect attempts.

Handle failures and uncertain operation outcomes even when a refresh succeeds. See Automatic Reconnect Limitations and Configuring Automatic Client Refresh Options.

Special Cases for Reconnected Connections

There are special cases that can occur when producer connections are refreshed:

- *Connections with a ClientID for Durable Subscribers* – If your Reconnect Policy field is set to **None** or **Producer**, and a JMS Connection has a Client ID specified at the time of a network/server failure, then the Connection will not be automatically refreshed. The reason for this restriction is backward compatibility, which avoids breaking existing JMS applications that try to re-create a JMS Connection with the same connection name after a failure. If implicit failover also occurs on a network failure, then the application's creation of the connection will fail due to a duplicate ClientID.

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

> **Note**
>
> For information on how a consumer connection with a ClientID behaves, see Consumer Connections with a ClientID for Durable Subscriptions.

- *Closed Objects Are Not Refreshed* – When the application calls `jakarta.jms.Connection.close()`, `jakarta.jms.Session.close()`, etc., that object and it descendents are not refreshed. Similarly, when the JMS client is told its Connection has been administratively destroyed, it is not refreshed.
- *Connection with Registered Exception Listener* – If the JMS Connection has an application ExceptionListener registered on it, that ExceptionListener's `onException()` callback will be invoked even if the connection is implicitly refreshed. This notifies the application code of the network disconnect event. The JMS client application code might normally call `connection.close()` in `onException`; however, if it wants to take advantage of the reconnect feature, it may choose not to call `connection.close()`. The registered ExceptionListener is also migrated transparently to the internally refreshed connection to listen for exceptions on the refreshed connection.
- *Multiple Connections* – If there are multiple JMS Connections created off the same ConnectionFactory object, each connection will behave independently of the other connections as far as the reconnect feature is concerned. Each connection will have its own connection status, its own connection retry machinery, etc.

Reconnected Session Objects

As described in Reconnected Connection Objects, JMS Session objects are refreshed when their associated JMS connection gets refreshed . Session states, such as acknowledge mode and transaction mode, are preserved across each refresh occurrence. The same session object can be used for calls, like `createProducer(Destination)`, after a refresh. See Example 14-1 and Example 14-2.

Special Cases for Reconnected Sessions

These sections discuss special cases that can occur when Sessions are reconnected.

- Transacted sessions with pending work — When a local JMS session is refreshed with pending transactional work, its first commit can throw jakarta.jms.TransactionRolledBackException. Handle this together with other messaging and transaction-completion exceptions. A refreshed connection does not guarantee that subsequent sends or receives succeed.

  After a confirmed rollback, retry the logical unit of work only when doing so is safe for all effects of that work. Do not treat every commit exception as proof of rollback or blindly repeat external effects that did not participate in the JMS transaction.
- Pending unacknowledged messages — A refreshed session with pending unacknowledged work can report weblogic.jms.common.LostServerException on acknowledgement. The exception indicates a connection-loss condition; handle possible redelivery and do not assume that the acknowledgement removed the earlier deliveries.

Reconnected MessageProducer Objects

As described in Reconnected Connection Objects, JMS `MessageProducer` objects are refreshed when their associated JMS connection gets refreshed . If producers are non-anonymous, that is, they are specific to a destination object (standalone or distributed

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

destination), then the producer's destination is also implicitly refreshed, as described in Reusable Destination Objects. If a producer is anonymous, that is not specific to a destination object, then the possibly stale destination object specified on the producer's `send()` operation is implicitly refreshed. See Example 14-1.

Special Case for Distributed Destinations

It is possible that a producer can send a message at the same time that a distributed destination member becomes unavailable. If WebLogic JMS can determine that the distributed destination member is not available, or was not available when the message was sent, the system will retry sending the message to another distributed member. If there is no way to determine if the message made it through the connection all the way to the distributed member before it went down, the system will not attempt to resend the message because doing so may create a duplicate message. In that case, WebLogic JMS will throw an exception. It is up to the application to catch that exception and decide whether or not to resend the message.

Configuring Automatic Failover for JMS Consumers

When the deprecated mechanism is active, an effective ReconnectPolicy of all allows eligible MessageConsumer objects to participate in connection refresh. Calling WLConnection.setReconnectPolicy("all") alone does not enable automatic reconnection in the default configuration of this release. See Automatic JMS Client Failover.

Explicitly enabling automatic refresh of consumers also refreshes connections with a configured client ID for a durable subscriber, as described in Consumer Connections with a ClientID for Durable Subscriptions. However, consumer refresh does not include `QueueBrowser` clients, which are never refreshed, as described in Automatic Reconnect Limitations.

Sample Consumer Client Code

Example 14-2 is pseudocode illustrating consumer resource use and failure handling. It is not executable Java and does not implement a complete recovery loop. The deprecated automatic-reconnect mechanism is disabled by default.

Example 14-2 Sample JMS Client Code for Message Consumption

```
1. Create a JNDI context using the application's connection settings.
2. Look up the connection factory and destination.
3. Create a connection and session using the required acknowledgement
   or transaction mode.
4. Register an exception listener that signals the application's recovery coordinator.
5. Create the consumer; register a message listener for asynchronous delivery.
6. Start the connection.
7. Receive and process messages, completing acknowledgements or transactions
   according to the selected mode.
8. On failure, coordinate cleanup and recreation of the failed resources.
   Restore the intended subscription identity and handle possible redelivery.
9. Close the connection and JNDI context on completion or failure.
```

Use the application's recovery logic to restore consumer resources after a connection failure. Do not assume that calling setReconnectPolicy("all") activates recovery in the default configuration. See Client Resiliency Best Practices in Administering JMS Resources for Oracle WebLogic Server.

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

Configuring Automatic Client Refresh Options

The following deprecated settings describe the legacy reconnect mechanism. They do not enable automatic reconnection in this release's default configuration. The defaults shown are descriptor defaults, not a promise that reconnect is active.

Table 14-1 Automatic JMS Client Reconnect Options

MBean Attribute	Legacy values or default	Description
Reconnect Policy `ReconnectPolicy`	• `none` • `producer` (descriptor default) • `all`	When the deprecated mechanism is active, controls which eligible connection, session, producer and consumer objects participate in refresh. QueueBrowser objects are not refreshed. The current default effective policy is none because automatic reconnection is disabled.
Reconnect Blocking Time `ReconnectBlockingMillis`	60000 (60 seconds)	Limits a synchronous caller's wait for reconnect processing when the deprecated mechanism is active. A value of 0 disables that wait; -1 permits an indefinite wait. It is not a general end-to-end operation timeout.
`TotalReconnectPeriodMillis`	`-1`	Controls the total period during which the deprecated mechanism attempts reconnection. The default value of -1 permits retries without a configured time limit. Closing the connection stops reconnect attempts.

> **Note**
>
> The configuration options in the table are not supported in the WebLogic Remote Console, as these options are for an already deprecated feature.

For more information about these parameters, see `ClientParamsBean` in the *MBean Reference for Oracle WebLogic Server*.

Common Cases for Reconnected Consumers

This section describes the common scenarios when refreshing synchronous and asynchronous consumers.

Synchronous Consumers

Use MessageConsumer.receive(), receive(timeout), or receiveNoWait() according to the application's message-waiting requirements. In normal operation, receive(timeout) can return null when no message arrives within the requested interval, and receiveNoWait() can return null when no message is immediately available. A connection failure can instead cause a JMSException.

When the deprecated automatic-reconnect mechanism is active, reconnect processing can add waiting before the receive operation proceeds. Do not use receive(timeout) as a strict end-to-end bound on a call that encounters reconnection, or assume that receiveNoWait() bypasses all reconnect processing. ReconnectBlockingMillis controls reconnect waiting rather than the normal wait for a message.

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

After recovery, consumers must handle possible redelivery. Handle synchronous exceptions as well as asynchronous connection notifications. For acknowledgement and transaction completion after legacy refresh, see Special Cases for Reconnected Sessions.

Asynchronous Consumers

In the context of a reconnect, the behavior for asynchronous consumers will be governed by the setting on the Total Reconnect Period property. The JMS consumer's registered message listener's `onMessage()` method will continue to be invoked if the reconnect framework is able to successfully re-establish a connection within the Total Reconnect Period setting after a connection failure. If the user explicitly calls a `close()` on the JMS Connection (or on the JMS Session corresponding to the asynchronous Consumer), then the reconnect framework will not invoke any further `onMessage() calls` for that Consumer. The `onMessage()` should expect post recover behavior (like redelivered messages) if the Connection ExceptionListener's `onException` is invoked with a `LostServerException`.

Special Cases for Reconnected Consumers

These sections discuss special cases that can occur when consumers are refreshed.

Consumers of Distributed Destinations

A consumer attached to a distributed destination receives from a selected physical member. If a consumer is recreated during recovery, member selection can choose the same member or a different member, depending on the destination and configuration. Do not rely on recovery to preserve member selection.

Create durable subscriptions on physical distributed-topic members, or use MDBs or documented helpers to manage those member subscriptions. Direct creation of a durable consumer on the logical distributed-topic object is not supported. See Considerations When Using Standard Shared Subscriptions.

Message-Driven EJBs

Message-driven EJBs (MDBs) are a special sub case of asynchronous consumers that have their own behavior requirements and their own refresh framework. As such, MDBs are not expected to participate in MessageConsumer refreshes, and are not expected to be affected in any other way by the JMS client reconnect framework.

Consumer Connections with a ClientID for Durable Subscriptions

When the deprecated mechanism is active, a connection with a client ID is eligible for refresh only when the effective reconnect policy includes consumers. Successful recovery of a durable consumer depends on the availability of its durable subscription. Preserve its identity when recreating it, and handle exceptions and possible redelivery.

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

> **Note**
>
> A Restricted client ID can delay reconnection while the previous connection's registration is being released. Client ID Policy and subscription identity affect recovery; see Shared Subscriptions and Client ID Policy.
>
> Durable subscriptions on different physical distributed-topic members remain separate subscriptions, even when their names and client IDs match. Recovery does not move a member's retained messages into another member's subscription.

Non-Durable Subscriptions and Possible Missed Messages

A non-durable subscription does not retain messages after the subscription ceases to exist. Messages published while a failed subscription is absent can be missed when it is recreated. For a shared non-durable subscription, failure of one consumer does not by itself remove the subscription if other consumers remain attached. Recovery must account for both the subscription's lifetime and the messages eligible for that subscription.

Duplicate Messages

Connection recovery can cause previously delivered messages to be delivered again, even when the application did not explicitly call recover(). Design processing for the acknowledgement and transaction mode in use, including its redelivery possibilities. Recovery does not establish an exactly-once processing guarantee.

A non-durable RDT consumer that reconnects to a different member can also encounter a replicated copy of a message it received earlier. Such a copy need not be marked as redelivered to that new subscription. Do not rely solely on JMSRedelivered to identify duplicate business work.

Variations Due to Acknowledge Modes

The acknowledgement or transaction mode determines how an application completes its messaging work. After legacy refresh, an acknowledgement can fail when the session had pending unacknowledged work. Locally transacted sessions instead complete work through commit or rollback. See Special Cases for Reconnected Sessions.

Reconnecting with Migrated JMS Destinations In a Cluster

Migrating a JMS server moves the destination service; it does not migrate application-side Connection, Session, or MessageConsumer objects. A client can receive a synchronous JMSException or an asynchronous ExceptionListener notification when its connection fails. Coordinate cleanup and recreation of the affected client resources after the service becomes available, preserving the intended durable-subscription identity. See Client Resiliency Best Practices in Administering JMS Resources for Oracle WebLogic Server.

Explicitly Disabling Automatic Failover on JMS Clients

Automatic JMS client reconnection is disabled by default in this release. The following settings apply only to environments where the deprecated mechanism is active.

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

Programmatically

For an existing WebLogic JMS Connection named con, the following legacy API call requests a policy of none. The enclosing method must have created con and must handle connection-creation and messaging errors.

```
((weblogic.jms.extensions.WLConnection) con).setReconnectPolicy("none");
```

For more information about the `setReconnectPolicy` method, see the `weblogic.jms.extensions.WLConnection` API.

Administratively

Where the deprecated reconnect mechanism is active, set the connection factory's ReconnectPolicy attribute to none. Connection-factory changes apply to newly created connections. These settings are not exposed by WebLogic Remote Console.

For more information about the other JMS connection factory client parameters, see `ClientParamsBean` in the *MBean Reference for Oracle WebLogic Server*.

Best Practices for JMS Clients Using Automatic Failover

Oracle recommends the following best practices for JMS clients when using the Automatic JMS Client Reconnect feature:

Always Catch Exceptions

There are some connection failure scenarios where the result of a message operation is undetermined and WebLogic Server throws an exception. Your application must deal with the exception appropriately. See the following:

- Automatic Reconnect Limitations
- Special Cases for Reconnected Sessions
- Special Case for Distributed Destinations

Use Transactions to Group Message Work

Use a local JMS transaction to group sends and receives on one session, or an appropriate JTA transaction to coordinate enlisted resources. Only operations participating in that transaction share its completion outcome. Handle completion exceptions and resolve uncertain outcomes before retrying work; a server failure does not by itself prove that an earlier commit failed.

See Using Transactions with WebLogic JMS.

> **Note**
>
> Transaction recovery can continue after a server or client connection is restored. A local JMS TransactionRolledBackException reports rollback, but other completion failures can require further investigation. For JTA, follow the transaction manager's recovery and heuristic-outcome guidance.

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

JMS Clients Should Always Call the close() Method

As a best practice, your applications should not rely on the JVM's garbage collection to clean up JMS connections because the JMS automatic reconnect feature keeps a reference to the JMS connection. Therefore, always use `connection.close()` to clean up your connections. Also consider using a `finally` block to ensure that your connection resources are cleaned up. Otherwise, WebLogic Server allocates system resources to keep the connection available.

For more information about closing JMS client connections, see Best Practice: Always Close Failed JMS ClientIDs.

Manually Migrating JMS Data to a New Server

WebLogic JMS uses the migration framework to allow WebLogic JMS to respond properly to migration requests and bring a WebLogic JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure.

A JMS server can move between eligible WebLogic Server hosts when its targeting, migration policy, persistent store, and recovery configuration support that move. Use the service-migration procedure appropriate to that configuration.

Before activating the recovered service, preserve access to its existing persistent-store data and ensure that the previous host cannot concurrently activate the same service or write to the same store. The following table summarizes recovery checks; it does not replace the linked migration and persistent-store procedures.

> **Note**
>
> There are special considerations when you migrate a service from a server instance that has crashed or is unavailable to the Administration Server. If the Administration Server cannot reach the previously active host of the service at the time you perform the migration, see Migrating a Service From an Unavailable Server in *Administering Clusters for Oracle WebLogic Server*.

Table 14-2 Migration Task Guide

If Your JMS Application Uses	Perform the Following Task
Persistent messaging—JDBC Store	Ensure that the recovery host can access the existing JDBC store's database and backing table through the correctly configured data source. If the database endpoint changes, update the data-source configuration as required. A database that remained available still requires the JMS service, store targeting, access and ownership checks.
Persistent messaging—File Store	Ensure that the recovery host can access the existing store's data files through the configured store directory. If files must be copied, first ensure that the store is inactive and cannot restart during the copy, and preserve a consistent complete set of its files. Configure the recovery location according to the persistent-store recovery procedure before activating the service. See Using Custom File Stores and File Locations in *Administering the WebLogic Persistent Store*.

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

Table 14-2 (Cont.) Migration Task Guide

If Your JMS Application Uses	Perform the Following Task
Transactions	Coordinate JMS store recovery with transaction recovery. The Transaction Recovery Service needs access to the failed server's transaction-log records to resolve outstanding transactions. For detailed instructions about recovering transactions from a failed server, see Transaction Recovery After a Server Fails in *Developing JTA Applications for Oracle WebLogic Server*.

> **Note**
>
> Recovery of a persistent store containing many messages can require substantial memory. If startup fails because of insufficient memory, investigate the failure and review JVM heap sizing and available host memory before retrying. Recovery requirements depend on the stored workload and configuration; do not size the heap solely in proportion to message count.

See Starting and Stopping Servers: Quick Reference. For information about recovering a failed server, refer to Avoiding and Recovering From Server Failure in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

For more information about defining migratable services, see Service Migration in *Administering Clusters for Oracle WebLogic Server*.

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

15

Understanding WebLogic JMS Security

Learn how to secure WebLogic JMS resources using thread-based and object-based security models.

Securing WebLogic JMS Resources

WebLogic JMS enables you to secure JMS resources by restricting access to JMS destinations.

The default root-level JMS security policy permits the Everyone group. Configure more specific policies on JMS modules, destinations, or destination operations to restrict access. This JMS policy does not bypass authentication, JNDI access controls, or administrative permissions. Ensure that the intended users satisfy the applicable policy conditions. See Overview of Securing WebLogic Resources for security roles and policies, and Java Messaging Service (JMS) Resources for JMS policy targets and operations.

JMS Security Terminology

WebLogic JMS uses either object-based security (OBS) or thread-based security to determine which user is checked when accessing a secured WebLogic JMS destination.

Understand some common terminology used in the context of JMS security before exploring the difference between the two security approaches:

- Subject: A security identity represented by a set of principals and, where applicable, credentials.
- Principal: An identity associated with a subject, such as a user or group.
- Credentials: Information used to authenticate an identity, such as a password. These examples use a user name and password supplied through the applicable JNDI or JMS API.

Thread-based security implies that the subject that a secured WebLogic JMS destination checks is implicitly derived from the current caller's thread. Object-based security implies that the subject is implicitly derived from a subject stored in the object the caller is using to make its JMS call. In general, WebLogic security is thread based. The following sections explore both approaches.

Understanding Thread-Based Security on Clients and Servers

By default, access to secured WebLogic JMS resources leverages thread-based security. This gives WebLogic JMS security behavior parity with Jakarta EE's general security model for EJBs, web applications, and RMI.

With thread-based security, WebLogic JMS send and consume authorization uses the calling thread's subject. Supplying a user name and password to ConnectionFactory.createConnection() or ConnectionFactory.createContext() does not, by itself, replace that subject for subsequent thread-based JMS operations. Object-based security and the server-managed patterns described later in this chapter use different identity-selection rules.

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

Thread-Based Security for Server Applications

For server-side applications, there are multiple ways to set EJB and Web application thread's subject or role. See Securing Enterprise JavaBeans (EJBs). To override thread-based JMS security checking behavior for server-side WebLogic JMS send and consume calls, see Enabling Object-Based Security on Server Applications.

Thread-Based Security for Client Applications

For standalone clients using weblogic.jndi.WLInitialContextFactory, creating an InitialContext with credentials establishes the context's subject on the calling thread. Specify the credentials using Context.SECURITY_PRINCIPAL and Context.SECURITY_CREDENTIALS. Close the context on the thread that created it; when contexts are nested, close them in reverse creation order to restore the previous subject correctly.

The following method-body fragment uses caller-supplied url, username, and password values. Obtain credentials through the application's configuration mechanism; do not embed real passwords in source.

```
java.util.Hashtable<String, Object> env = new java.util.Hashtable<>();
env.put(javax.naming.Context.PROVIDER_URL, url);
env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
env.put(javax.naming.Context.SECURITY_PRINCIPAL, username);
env.put(javax.naming.Context.SECURITY_CREDENTIALS, password);
javax.naming.InitialContext ic = new javax.naming.InitialContext(env);
try {
    // Perform lookups and thread-based JMS operations here.
} finally {
    ic.close();
}
```

If a client creates an `InitialContext` object without specifying credentials, then:

- The subject already on the current thread is unchanged.
- If there is no subject on the thread, then it is assumed to be an *anonymous* subject.

If a client program needs to transfer a thread's subject to a different thread than the thread used to create an `InitialContext` object, the client program can use security APIs to store the current subject of the thread and then subsequently use this cached subject in a different thread.

Capture the authenticated subject with weblogic.security.Security.getCurrentSubject() while the intended context is active. Pass that captured subject to the helper on the thread that performs the operation. The helper propagates PrivilegedActionException; its cause is the checked exception thrown by the action, not necessarily an authentication error.

```
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import javax.security.auth.Subject;
import weblogic.security.Security;

public final class SubjectActionExample {
    public static Object doSomethingAsSubject(Subject subject)
```


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

```
            throws PrivilegedActionException {
        return Security.runAs(subject, (PrivilegedExceptionAction<String>) ()
-> {
            // Perform the operation using this subject.
            return "OK";
        });
    }
}
```

This example code pattern can also be used to switch the subject on a thread for the use case where one single JMS client communicates with two domains at the same time, see Programming Pattern for a Single JMS Client Communicating With Two WebLogic Domains.

Sometimes it is useful to get an *anonymous* subject:

```
javax.security.auth.Subject anon = new javax.security.auth.Subject();
```

To override thread-based JMS security checking behavior for client-side WebLogic JMS send and consume calls, see Enabling Object-Based Security on Clients.

Understanding Object-Based Security

WebLogic JMS clients can optionally use a simpler security model called object-based security (OBS) instead of thread-based security. This option was introduced in WebLogic 12.2.1.3 and is useful for multithreaded clients which otherwise need extra code to transfer thread-based security subjects between threads.

The following sections explain how to enable object-based security:

Enabling Object-Based Security on Clients

Enabling object-based security (OBS) causes message send and consume security checks to be based on credentials specified during JMS client initialization instead of on the calling thread's subject.

Enabling OBS requires using an OBS JNDI initial context. Any WebLogic JMS senders or consumers that are created using an OBS connection factory that is obtained from an OBS initial context will, by default, implicitly use the credential that is associated with the OBS initial context instead of the subject that is associated with the current sender or consumer thread. In addition, if a user name and password credential is passed as parameters to a standard JMS `createConnection()` or `createContext()` call on an OBS connection factory, then this new credential supersedes the credential that is associated with the OBS initial context and the new credential will be used for sends or consumes on that connection or JMS context.

Steps to enable OBS on a JMS client's senders and consumers:

1. Create a `javax.naming.InitialContext` object:
    a. Specify a `Context.INITIAL_CONTEXT_FACTORY` property with string value `weblogic.jms.WLInitialContextFactory` instead of `weblogic.jndi.WLInitialContextFactory`. This returns an OBS initial context, and is the only step required for the majority of applications that want to use JMS client OBS.
    b. (Optional) Specify user name and password credentials using the standard JNDI `Context.SECURITY_PRINCIPAL` and `Context.SECURITY_CREDENTIALS` properties (same as what you do for a non OBS context). This will become the default OBS credential

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

that is associated with the OBS initial context. If these properties are not specified, then the credential associated with the OBS initial context is determined by the `weblogic.jndi.securityPolicy` setting.

c. (Optional) Specify an initial context property named `weblogic.jndi.securityPolicy` with string value `ObjectBased` or `ObjectBasedHybrid`. This fine tunes behavior when no user name and password credential is specified using the initial context `Context.SECURITY_PRINCIPAL` and `Context.SECURITY_CREDENTIALS` properties.

   - ObjectBased (the default): If no user name is supplied when the OBS initial context is created, use an anonymous subject for its JNDI lookups and, by default, for JMS operations through factories obtained from that context.
   - ObjectBasedHybrid: If no user name is supplied when the OBS initial context is created, capture the current thread's subject for its subsequent JNDI lookups and, by default, for JMS operations through factories obtained from that context.

2. Use the OBS initial context created in step (1) to look up WebLogic JMS connection factories (same as you look up a connection factory for a non OBS context), which will then implicitly be OBS JMS connection factories. Any JMS senders or consumers that are created using an OBS JMS connection factory will use OBS.
3. (Optional) Override the OBS credential associated with the OBS JMS connection factory by passing a user name and password into the JMS standard `createConnection()` or `createContext()` call that is used to create a JMS connection or context.

Object-Based Security Limitations on Clients

Listed below are some of the limitations of OBS:

- Use lookup(String) to obtain supported WebLogic JMS connection factories and destinations, and close() to release the OBS context. lookup(Name) and unsupported naming operations throw javax.naming.OperationNotSupportedException. OBS lookups do not support JMS resources referenced through a foreign JMS server. Use a separate non-OBS context for naming operations or resource types that the OBS context does not support.
- An OBS initial context is supported only on WebLogic clients and is not supported on WebLogic Servers. An exception is thrown when attempting to use such a context in combination with WebLogic JMS server facilities like bridges, MDBs, or resource references. This restriction exists because these server-side JMS facilities already have their own security handling that provides similar semantics to OBS.
- When an OBS initial context is created, the initial context's user is not placed on the current thread. This differs from a `weblogic.jndi.WLInitialContextFactory` context.

Enabling Object-Based Security on Server Applications

Learn how to enable object-based security (OBS) for inbound and outbound JMS applications:

Object-Based Security for Inbound JMS Applications

Server applications that make inbound WebLogic JMS calls, such as Message Driven Beans, are implicitly object based. Credentials or roles for these applications to access incoming JMS messages are supplied in one of the following ways:

- Supplied with the application itself.

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

> **Note**
>
> Oracle does not recommend using this method.

- Defined on the service itself (Messaging Bridges allow you to configure user name or password).

> **Note**
>
> Oracle does not recommend using this method.

- Just as for the outbound case, specified using a foreign JMS Server in a JMS system resource module that maps a JMS resource into JNDI.

As a best practice, use the foreign JMS Server method for Message Driven Beans, Messaging Bridges, and outbound JMS, as this:

- Centralizes credential configuration outside application code. The relevant credential attributes are dynamically configurable; existing connections and resource pools remain subject to their refresh and lifecycle rules.
- Applies to almost all inbound and outbound use cases so it is useful as a way to centrally manage your JMS credentials.

See FAQs: Integrating Remote JMS Providers and Enhanced Support for Using WebLogic JMS with EJBs and Servlets.

Object-Based Security for Outbound JMS Applications

Server applications that make outbound WebLogic JMS calls can achieve an object-based security pattern that supersedes the current security subject on the current thread. For this to work, ensure the following:

1. Configure a foreign JMS server and a foreign connection-factory mapping from the remote factory's JNDI name to a local JNDI name. Configure the foreign server's JNDI properties and JNDI credential for the remote naming lookup. If connection authentication requires a user name and password, configure those separately on the foreign connection factory. Ensure that the resulting identity has the required remote permissions.
2. In the application code, use a JMS resource reference or inject a JMS context that references the local JNDI name of the JMS connection factory.

> **Note**
>
> It is a general best practice for server applications to use resource references or JMS context injection to reference a JMS connection factory regardless of whether you need an object-based security pattern or a thread-based security pattern.
>
> With this server-managed pattern, WebLogic uses the configured identity when creating and using the managed JMS resources. JNDI lookup credentials and JMS connection credentials serve different purposes; a successful naming lookup alone does not establish authorization for sends or receives.

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

Understanding Cross-Domain Security

By default, WebLogic Server security is thread-based, which means operations are performed as the "user" that is associated with the current thread. See Understanding Thread-Based Security on Clients and Servers. When an application uses JMS to communicate with multiple domains, the application needs to ensure that the correct user is used when it communicates with each of the domains that are involved.

Application authentication to a remote JMS provider and security for internal communication between WebLogic domains are separate concerns. Configure the inter-domain security required by the participating services and topology. When using WebLogic Cross-Domain Security, enable it and configure the required cross-domain users and credential mappings for the participating domain pairs. See Configuring Cross-Domain Security in Administering Security for Oracle WebLogic Server. The following Cross-Domain Security Guidelines identify service-specific considerations.

> **Note**
>
> The examples provided are example code for illustrating security patterns, not working code that conforms to all JMS coding best practices. See Configuration Best Practices.

Cross-Domain Security Guidelines

Follow these guidelines while configuring cross-domain security:

- If your Message-Driven Bean (MDB) deployment and the JMS destination that the MDB listens on, are in different WebLogic domains, you need to consider configuring cross-domain security between the two domains in combination with a foreign JMS server. For more details, see Using MDBs With Cross Domain Security in *Developing Message-Driven Beans for Oracle WebLogic Server*.
- For applications or messaging bridges participating in global transactions across WebLogic domains, configure compatible transaction-security settings in all participating domains. The JTA configuration supports Cross Domain Security or the applicable Security Interoperability Mode; follow the JTA guide for the required combination. This applies to messaging bridges using exactly-once QoS across WebLogic domains. See Configuring Cross Domain Security in Developing JTA Applications for Oracle WebLogic Server.
- If your application in one WebLogic domain accesses WebLogic Server JMS distributed destinations in another WebLogic domain, you need to consider configuring cross-domain security between the two domains. In addition, the application should use a foreign JMS server in combination with a standard Jakarta EE resource reference to reference the remote destination.
- If your application uses WebLogic Server JMS store-and-forward to forward messages from one WebLogic domain to another WebLogic domain, you need to consider configuring cross-domain security between the two domains. See SAF and Cross Domain Security in *Administering the Store-and-Forward Service for Oracle WebLogic Server*.
- If a single JMS client communicates with multiple domains at the same time, the application code may need to manage switching users back and forth corresponding to the

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

domain that the application talks to using the same thread. See the `runAs helper` method described in Thread-Based Security for Client Applications.

- If using a foreign JMS server in a cross-domain scenario without combining this feature with a Jakarta EE resource reference, MDB, or messaging bridge, a built-in helper API can be used to correctly handle security credential propagation.

> **Note**
>
> It is a best practice to use a Jakarta EE resource reference, MDB, or messaging bridge instead of a helper API.

Programming Pattern for a Single JMS Client Communicating With Two WebLogic Domains

A single JMS client that is not running on a WebLogic Server may need to communicate with two (or more) WebLogic domains at the same time while specifying a valid thread-based security subject for each.

The following pseudocode illustrates the identity-selection error. A thread-based client must use the appropriate subject for each domain.

```
Create a thread-based naming context for domain 1.
Create the request consumer while domain 1's subject is current.
Create a thread-based naming context for domain 2 on the same thread.
Attempt to receive from domain 1 while domain 2's subject is current.
```

The receive can fail if the current subject is not authorized in domain 1. The problem is the selected identity, not simply the fact that two domains are involved.

Capture each domain's authenticated subject while its naming context is active, and use Security.runAs for operations in that domain. Creating JMS resources under a subject does not eliminate the need to use the appropriate subject for subsequent thread-based JMS operations. See Thread-Based Security for Client Applications. The following standalone-client example handles one TextMessage request and sends an illustrative text response. It uses AUTO_ACKNOWLEDGE and does not make request receipt and response sending atomic; it is not an exactly-once forwarding solution. The caller supplies correctly configured naming environments and JNDI names. Messaging failures propagate to the caller.

```
import jakarta.jms.ConnectionFactory;
import jakarta.jms.Destination;
import jakarta.jms.JMSConsumer;
import jakarta.jms.JMSContext;
import jakarta.jms.TextMessage;
import java.security.PrivilegedExceptionAction;
import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.security.auth.Subject;
import weblogic.security.Security;

public final class TwoDomainExample {
    // env1 and env2 contain independently configured JNDI URLs and
credentials,
```


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

```
    // and use weblogic.jndi.WLInitialContextFactory (thread-based security).
    public static void exchangeOnce(
            Hashtable<String, Object> env1, Hashtable<String, Object> env2,
            String requestFactoryName, String requestDestinationName,
            String responseFactoryName, String responseDestinationName)
            throws Exception {
        InitialContext naming1 = new InitialContext(env1);
        try {
            Subject subject1 = Security.getCurrentSubject();
            ConnectionFactory factory1 =
                    (ConnectionFactory) naming1.lookup(requestFactoryName);
            Destination requestDestination =
                    (Destination) naming1.lookup(requestDestinationName);
            try (JMSContext jms1 = factory1.createContext();
                 JMSConsumer consumer =
jms1.createConsumer(requestDestination)) {
                InitialContext naming2 = new InitialContext(env2);
                try {
                    Subject subject2 = Security.getCurrentSubject();
                    ConnectionFactory factory2 =
                            (ConnectionFactory)
naming2.lookup(responseFactoryName);
                    Destination responseDestination =
                            (Destination)
naming2.lookup(responseDestinationName);
                    try (JMSContext jms2 = factory2.createContext()) {
                        String requestText = (String) Security.runAs(subject1,
                                (PrivilegedExceptionAction<String>) () -> {
                                    TextMessage message =
                                            (TextMessage)
consumer.receive(1000);
                                    return message == null ? null :
message.getText();
                                });
                        if (requestText != null) {
                            Security.runAs(subject2,
                                    (PrivilegedExceptionAction<Void>) () -> {

jms2.createProducer().send(responseDestination,
                                                "Response: " + requestText);
                                    return null;
                                });
                        }
                    }
                } finally {
                    naming2.close();
                }
            }
        } finally {
            naming1.close();
        }
    }
}
```


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

Programming Patterns for Using a Foreign JMS Server Between Two WebLogic Domains

> **Note**
>
> Use the managed credential handling provided by Jakarta EE resource references, messaging bridges, or MDBs where applicable. The manual pattern below is for applications that do not use those facilities. Configure application permissions and any required inter-domain service security separately.

A foreign JMS server maps local JNDI names to remote JMS resources. Credentials used to resolve the foreign JNDI reference apply to that lookup; do not assume that they become the calling thread's identity for later JMS operations. A direct lookup outside managed resource-reference, MDB, or messaging-bridge handling can therefore require explicit identity management. Subsequent operations can fail if the effective subject is not authorized for the remote operation, even when the lookup credentials are correct.

The following bounded standalone-client example waits for an initial destination-availability notification before using RegistrationHandle.runAs(). The supplied JNDI environment identifies the domain containing the foreign mappings; the destination and connection-factory names must use the same foreign-server credential configuration. The 30-second wait is an example application timeout, not a product default. An availability notification establishes initial readiness only; the destination can become unavailable afterward, and subsequent operations can fail. For ongoing availability and recovery handling, see Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API.

For this example, the identity established by the foreign server's JNDI credentials must be authorized to create the remote JMS context and send to the destination. The sample does not pass separate JMS connection credentials or automatically apply a different username and password configured on the foreign connection factory.

```
import jakarta.jms.ConnectionFactory;
import jakarta.jms.Destination;
import jakarta.jms.JMSContext;
import jakarta.jms.JMSProducer;
import java.security.PrivilegedExceptionAction;
import java.util.Hashtable;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;
import javax.naming.InitialContext;
import weblogic.jms.extensions.DestinationAvailabilityListener;
import weblogic.jms.extensions.DestinationDetail;
import weblogic.jms.extensions.JMSDestinationAvailabilityHelper;
import weblogic.jms.extensions.RegistrationHandle;

public final class ForeignDestinationExample {
    // Both names map through the same foreign server/security configuration.
    public static void send(Hashtable<String, Object> environment,
            String destinationName, String factoryName,
            String text, int messageCount) throws Exception {
```


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

```
        CountDownLatch ready = new CountDownLatch(1);
        AtomicReference<Exception> failure = new AtomicReference<>();
        DestinationAvailabilityListener listener =
                new DestinationAvailabilityListener() {
                    public void onDestinationsAvailable(String name,
                            List<DestinationDetail> destinations) {
                        if (!destinations.isEmpty()) ready.countDown();
                    }
                    public void onDestinationsUnavailable(String name,
                            List<DestinationDetail> destinations) {
                        // This bounded example does not implement recovery.
                    }
                    public void onFailure(String name, Exception exception) {
                        failure.compareAndSet(null, exception);
                        ready.countDown();
                    }
                };
        RegistrationHandle handle =
JMSDestinationAvailabilityHelper.getInstance()
                .register(environment, destinationName, listener);
        try {
            if (!ready.await(30, TimeUnit.SECONDS)) {
                throw new IllegalStateException("Timed out waiting for the
destination");
            }
            if (failure.get() != null) {
                throw new IllegalStateException("Destination registration
failed",
                        failure.get());
            }
            InitialContext naming = new InitialContext(environment);
            try {
                Destination destination = (Destination)
naming.lookup(destinationName);
                ConnectionFactory factory = (ConnectionFactory)
naming.lookup(factoryName);
                handle.runAs((PrivilegedExceptionAction<Void>) () -> {
                    try (JMSContext context = factory.createContext()) {
                        JMSProducer producer = context.createProducer();
                        for (int i = 0; i < messageCount; i++) {
                            producer.send(destination, text + i);
                        }
                    }
                    return null;
                });
            } finally {
                naming.close();
            }
        } finally {
            handle.unregister();
        }
    }
}
```


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

16

WebLogic JMS C API

Understand the requirements, design principles, security considerations and implementation guidelines needed to use the WebLogic JMS C API to create C clients that can access WebLogic JMS applications and resources.

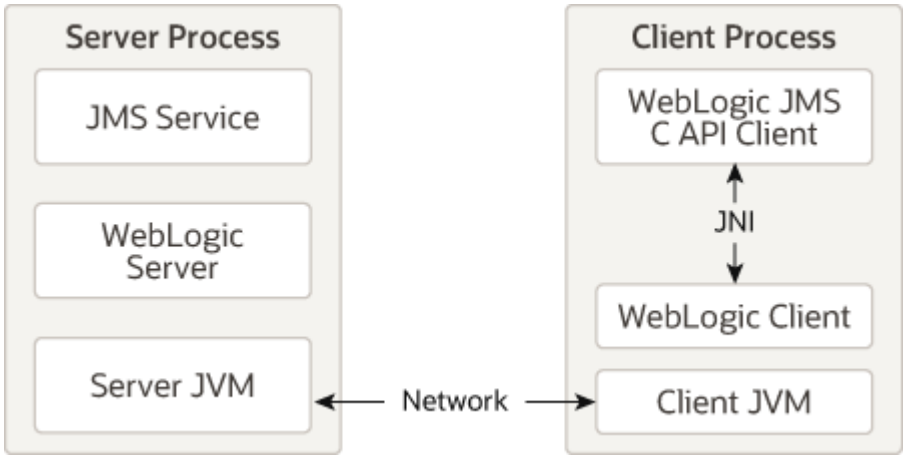
What Is the WebLogic JMS C API?

The WebLogic JMS C API is an application program interface that enables you to create C client applications that can access WebLogic JMS applications and resources.

The C API uses the Java Native Interface (JNI) to access the client-side Java messaging classes. See the Java Native Interface Specification and the following figure.

The WebLogic JMS C API provides a handle-based interface modeled on the JMS 1.1 programming model. See *JMS C API Reference for Oracle WebLogic Server* for the available functions and supported operations.

Figure 16-1 WebLogic JMS C API Client Application Environment



System Requirements

Understand the system requirements needed to use WebLogic JMS C API in your environment.

- A list of supported operating systems for the WebLogic JMS C API is available from the Oracle Fusion Middleware Supported System Configurations page. See Supported Configurations at *What's New in Oracle WebLogic Server*.
- A supported JVM for your operating system.
- An ANSI C compiler for your operating system.
- One of the following WebLogic clients to connect your C client applications to your JMS applications:

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

- The WebLogic Thin T3 Client jar (`wlthint3client.jar`). See Developing a WebLogic Thin T3 Client in *Developing Standalone Clients for Oracle WebLogic Server*.
- The WebLogic Install client (`weblogic.jar` file). See WebLogic Install Client in *Developing Standalone Clients for Oracle WebLogic Server*.

Design Principles

Understand the design principles for porting and developing applications for the WebLogic JMS C API.

Java Objects Map to Handles

The WebLogic JMS C API is handle-based to promote modular code implementation. This means that in your application you implement Java objects as handles in C code. The details of how a JMS object is implemented is hidden inside a handle. However, unlike in Java, when you are done with a handle, you must explicitly free it by calling the corresponding `Close` or `Destroy` methods. See Memory Allocation and Garbage Collection.

Thread Utilization

The C API follows the threading rules of the corresponding Java messaging objects. A JmsSession and its producers and consumers must be used according to the session's single-threaded programming model, including the restrictions on use while a message listener is running. A JmsConnection corresponds to the thread-safe jakarta.jms.Connection interface.

Synchronizing access does not transfer a thread's security subject or last-exception state. When operations run on another thread, establish the required security context on that thread and retrieve exception information on the thread where the failing call occurred. Observe the naming-context lifecycle requirements described in Understanding WebLogic JMS Security.

Exception Handling

> **Note**
>
> The WebLogic JMS C API uses integer return codes.

Exceptions in the WebLogic JMS C API are local to a thread of execution. The WebLogic JMS C API has the following exception types:

- `JavaThrowable` represents the class `java.lang.Throwable`.
- `JavaException` represents the class `java.lang.Exception`.
- `JmsException` represents the class `jakarta.jms.JMSException`. All standard subclasses of `JMSException` are determined by bits in the type descriptor of the exception. The type descriptor is returned with a call to `JmsGetLastException`.

Type Conversions

Java primitive types have fixed widths. Use the C types declared by the supplied JMS C API headers for the target platform. The following mappings require particular attention.

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

Integer (int)

Java int values map to JMS32I, a 32-bit signed value.

Long (long)

Java long values map to JMS64I, a 64-bit signed value.

Character (char)

The JMS C API character read and write functions use C short values to carry Java char values, which represent 16-bit UTF-16 code units. Use the declarations in the supplied headers.

String

`String` converts to `JmsString`.

`JmsString` supports two representations, selected by `stringType`:

- CSTRING: uniOrC.string points to a null-terminated string in JNI modified UTF-8. This encoding differs from standard UTF-8 for the null character and supplementary characters. See Modified UTF-8 Strings in the Java Native Interface Specification.
- UNISTRING: uniOrC.uniString.data points to native-order 16-bit UTF-16 code units suitable for JNI jchar data. For input, uniOrC.uniString.length specifies the number of bytes, which is twice the number of code units. Do not supply a serialized big-endian byte stream unless it has first been converted to the native representation.

For input, set `stringType` and the corresponding data field before calling the API.

When the `stringType` element of `JmsString` is used as output, the caller has the option to let the API allocate enough space for output using `malloc`, or you can supply the space and have the system copy the returned string into the provided bytes. If the appropriate field in the union (either string or data) is NULL, then the API allocates enough space for the output using `malloc`. It is the callers responsibility to free this allocated space using `free` when the memory is no longer in use. If the appropriate field in the union (string or data) is not NULL, then the `allocatedSize` field of `JmsString` must contain the number of bytes available to be written.

If there is not enough space in the string to contain the entire output, then `allocatedSize` sets to the amount of space needed and the API called returns `JMS_NEED_SPACE`. The appropriate field in the `JmsString` (either string or data) contains as much data as could be stored up to the `allocatedSize` bytes. In this case, the NULL character may or may not have been written at the end of the C string data returned. Example:

For example, to allocate 100 bytes for the string output from a text message, you would set the data pointer and the `allocatedSize` field to 100. The `JmsTextMessageGetText` API returns `JMS_NEED_SPACE` with `allocatedSize` set to 200. Call `realloc` on the original string to reset the data pointer and call the function again. Now the call succeeds, and you are able to extract the string from the message handle. Alternatively, you can free the original buffer and allocate a new buffer of the correct size.

Memory Allocation and Garbage Collection

Release resources explicitly when they are no longer needed. Java garbage collection does not replace closing JMS resources. In the C API, release each native handle using its

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

corresponding Close or Destroy function. Close functions invoke the associated Java JMS resource's close method and release the native handle. Destroy functions release the native handle and its Java references; do not assume that they invoke a Java close method. For example:

- The `jakarta.jms.Session` object has a `close` method so the WebLogic JMS C API has a `JmsSessionClose` function.
- The `jakarta.jms.ConnectionFactory` object does not have a `close` method so the WebLogic JMS C API has a `JmsConnectionFactoryDestroy` function.

> **Note**
>
> A handle that has been closed or destroyed should never be referenced again.

Closing Connections

JmsConnectionClose calls the underlying Java connection's close method, which closes its sessions, producers, and consumers. It releases the native connection handle, but does not free the separate native handles for those subordinate objects. Release every subordinate handle with its corresponding C API close function. A straightforward cleanup order is to close consumers and producers, then sessions, and finally the connection. Do not use a subordinate handle for messaging after its underlying resource has been closed, and do not reference a handle after its own close or destroy function has released it.

Helper Functions

The WebLogic JMS C API provides some helper functions that do not exist in WebLogic JMS. These helpers are explained fully in the *JMS C API Reference for Oracle WebLogic Server*. For example:

`JmsMessageGetSubclass` operates on a `JmsMessage` handle and returns an integer corresponding to the subclass of the message. In JMS, this could be accomplished using `instanceof`.

Security Considerations

A C client can supply a user name and password when creating its naming context.

Pass the credentials through the username and password arguments to JmsContextCreate. The C API supplies these values to the underlying initial context as javax.naming.Context.SECURITY_PRINCIPAL and javax.naming.Context.SECURITY_CREDENTIALS. Configure the identity with the required naming and JMS permissions. Authentication does not by itself authorize every messaging operation. Follow the thread-based subject and context-lifecycle guidance in Understanding WebLogic JMS Security.

Implementation Guidelines

Understand the limitations when you implement the WebLogic JMS C API.

- It does not support WebLogic Server JMS extensions, including XML messages.
- It does not support JMS Object messages.

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

- It creates an error log if an error is detected in the client. This error log is named `ULOG.mmddyy` (month/day/year). This log file is fully internationalized using the `NLSPATH`, `LOCALE`, and `LANG` environment variables of the client.
- Users who want to translate the message catalog can use the `gencat` utility provided on Windows or the `gencat` utility of the host platform. If the generated catalog file is placed according to the `NLSPATH`, `LOCALE`, and `LANG` variables, then the translated catalog will be used when writing messages to the log file.
- You can set the following environment variables in the client environment:
  - `JMSDEBUG`: Provides verbose debugging output from the client.
  - *`JMSJVMOPTS`*: Provides extra arguments to the JVM loaded by the client.
  - `ULOGPFX`: Configures the pathname and file prefix where the error log file is placed.

Client Packaging Requirements

You will need to include the JMS C API library and other files when you package the C application.

Include the following files along with a C application executable:

- A supported JVM for your operating system.
- If WebLogic Server is not installed on the machine that will run the application: the WebLogic JMS client jar(s) – usually the wlthint3client.jar. See Developing a WebLogic Thin T3 Client in *Developing Standalone Clients for Oracle WebLogic Server*.
- If the executable dynamically links the JMS C library, package the library for its operating system and processor architecture from the matching WebLogic installation. The executable, native library and JVM must have compatible architectures. For example, the packaging paths under WL_HOME include server/native/linux/x86_64/libjmsc.so for Linux x86-64 and server/native/win/x64/jmsc.dll for Windows x64. These examples are installation paths, not a certification list. Use Supported Configurations to confirm availability for the target release and platform.

Workarounds for Client Failure Thread Detach Issue

A C program that uses the JMS C client library may fail when its implicitly embedded JVM fails.

The JMS client failure could be related to a known, intermittent race-condition that occurs only with certain JVM products. The likelihood of failure can change based on the JVM version and patch level, operating system, and hardware combination. Specifically, the JMS C-Client library implicitly attaches C-threads to the JVM, but fails to detach them when it is done with them. The suggested workarounds are as follows:

- Before an application-created native thread exits, detach it from the JVM after its final JMS C API or JNI operation and after completing resource cleanup. Do not detach from a callback with Java methods still on its stack.
- Do not allow a C thread that has previously called into the JMS C-API to exit before the entire process exits.

The following C helper is for an application-created native thread immediately before it exits. It checks the single-JVM assumption and returns a JNI status. The caller must check that status. A thread that is already detached is treated as successfully detached; no additional detach operation is performed. After successful detachment, the thread must not make further JMS C API or JNI calls or reuse a cached JNIEnv pointer. See Sample C JNI Thread-Detach Helper.

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

Example 16-1 Sample C JNI Thread-Detach Helper

```
#include <jni.h>

/* Call only at the end of an application-created native thread's work. */
jint detach_before_thread_exit(void)
{
    JavaVM *vms[1];
    jsize count = 0;
    void *env = NULL;
    jint result = JNI_GetCreatedJavaVMs(vms, 1, &count);
    if (result != JNI_OK) {
        return result;
    }
    if (count != 1) {
        return JNI_ERR;
    }
    result = (*vms[0])->GetEnv(vms[0], &env, JNI_VERSION_1_2);
    if (result == JNI_EDETACHED) {
        return JNI_OK;
    }
    if (result != JNI_OK) {
        return result;
    }
    return (*vms[0])->DetachCurrentThread(vms[0]);
}
```

Compile this helper with the JNI headers for the selected JDK and platform. For example, in a Microsoft Visual C developer command prompt, with JAVA_HOME set to the JDK installation, compile detach.c using:

```
cl /W4 /c /I"%JAVA_HOME%\include" /I"%JAVA_HOME%\include\win32" detach.c
```

The /c option compiles the source to an object file; it does not link an executable. For an application that links directly to the JNI invocation API, link the object with the matching JDK's import library. If the selected Windows JDK supplies jvm.lib under its lib directory, add "%JAVA_HOME%\lib\jvm.lib" to the application's link inputs. Verify the library location and architecture for that JDK and follow the application's linkage model. Configure the runtime library search path for that JVM.

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

A

FAQs: Integrating Remote JMS Providers

Jakarta Messaging, Jakarta Transactions and the Java Naming and Directory Interface (JNDI) support integration with remote messaging providers. WebLogic Server provides managed resources and services that help applications use these APIs.

Understanding JMS and JNDI Terminology

Q. What is a remote JMS provider?

A. A remote JMS provider is a JMS server that is hosted outside a local standalone WebLogic Server or outside a WebLogic Server cluster. The remote JMS server can be a WebLogic or a non-WebLogic (foreign) JMS server.

Q. What is JNDI?

A. JNDI is the Java API for accessing naming and directory services. WebLogic Server uses it to expose resources such as connection factories, destinations, data sources and enterprise beans. Applications can look up resources by their configured names.

Cluster-wide bindings can make resources available through the cluster's naming service without requiring the application to know the hosting server. Lookup and use remain subject to binding scope, resource availability and security permissions.

Q. What is a JMS connection factory?

A. A JMS connection factory is a named entity resource stored in JNDI. Applications, message driven beans (MDBs), and messaging bridges lookup a JMS connection factory in JNDI and use it to create JMS connections. JMS connections are used in turn to create JMS sessions, producers, and consumers that can send or receive messages.

Q. What is a JMS client ID?

A. A client ID identifies a connection for subscription identity where the selected subscription API requires it. For the standard classic APIs, Session.createDurableConsumer() and Session.createDurableSubscriber() require a client ID; Session.createSharedDurableConsumer() can create a shared durable subscription without one. WebLogic's Restricted client-ID policy enforces exclusive registration within its applicable scope. Unrestricted policy permits multiple connections to use the same client ID. Subscription-sharing rules still apply; matching client IDs do not merge subscriptions on different physical distributed-topic members. See Developing Advanced Pub/Sub Applications.

Q. What is the difference between a JMS topic and a JMS queue?

A. Queue consumers compete for messages on a queue. For a topic, delivery is organized by subscriptions: each eligible subscription can receive a copy, and consumers of a shared subscription share delivery of that subscription's messages. Selection, acknowledgement, redelivery and expiration rules still apply.

Q. What is a topic subscription?

A. A subscription represents a topic's delivery relationship with one or more consumers. Messages published while it exists are considered for that subscription according to its selection rules. A subscription can be shared or unshared and durable or non-durable.

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

Creating a new subscription does not retrieve messages published before that subscription existed. Reopening an existing durable subscription can make messages previously retained for that subscription available.

Q. What is a non-durable topic subscriber?

A. A non-durable subscription exists while its consumer, or at least one consumer of a shared subscription, remains active. The standard shared non-durable APIs use a subscription name. WebLogic also supports policy-based sharing through traditional subscriber APIs, as described in Shared Subscriptions and Client ID Policy. A non-durable subscription does not preserve a backlog for consumption after that subscription has ended, and persistent delivery mode does not make the subscription durable.

Q. What is a durable subscriber?

A. A durable subscription retains eligible messages while it has no active consumer, subject to expiration and other applicable policies. Its identity includes the subscription name and any client ID required by the chosen API. For WebLogic distributed topics, subscriptions on different physical members remain separate. Persistent messages can be recovered with the durable subscription after a server restart, subject to the persistent store and recovery configuration. See Developing Advanced Pub/Sub Applications.

Understanding Transactions

Q. What is a transaction?

A. A transaction is a set of distinct application operations that must be treated as an atomic unit. To maintain consistency, all operations in a transaction must either all succeed or all fail. See Introducing Transactions in *Developing JTA Applications for Oracle WebLogic Server*.

Q. Why are transactions important for integration?

A. Integration applications often use transactions to ensure data consistency. For example, to ensure that a message is forwarded exactly-once, a single transaction is often used to encompass the two operations of receiving the message from its source destination and sending the message to the target destination. Transactions are also often used to ensure atomicity of updating a database and performing a messaging operation.

Q. What is a JTA/XA/global transaction?

A. Jakarta Transactions (JTA) coordinates global transactions that can include multiple enlisted resources, such as JMS and JDBC resources. XA provides the resource-manager interface used for distributed transaction coordination. UserTransaction is an API for application-managed transaction demarcation, not another transaction type. A transaction can be associated with a thread, suspended and resumed, or propagated through supported invocation paths.

Q. What is a local transaction?

A. A local JMS transaction groups sends and receives performed through one transacted session. It does not include JDBC work or operations through another session or JMS provider.

Q. How does JMS provide local transactions?

A. In the classic JMS API, create a transacted Session and use its commit or rollback methods. In the simplified API, use a locally transacted JMSContext where that mode is supported. Destination placement does not change the session transaction boundary. In Jakarta EE components, follow the container's transaction rules; a session participating in JTA is controlled by that JTA transaction.

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

Q. Are JMS local transactions useful for integration purposes?

A. Local JMS transactions are useful when the required atomic work fits within one transacted session. Use a global transaction when atomic completion must include separately enlisted resources, such as a database or another messaging provider.

Q. What is Automatic Transaction Enlistment?

A. Automatic enlistment means that the runtime associates a resource's work with the applicable JTA transaction without application code explicitly enlisting its XAResource. It requires a supported transaction integration and the appropriate connection-factory, resource and container configuration.

WebLogic JMS can participate through a transaction-enabled connection factory. WebLogic-managed JMS resource references and MDBs can provide transaction integration for supported third-party providers with the required XA capability. Messaging-bridge transaction behavior depends on its configured quality of service and adapter/provider support. A direct lookup of a foreign factory does not itself add WebLogic transaction management or pooling.

See Message-Driven EJBs, JMS resource-reference pools and Messaging Bridges for the supported configuration of each integration. Support for local transacted sessions alone does not establish XA/global transaction support.

How to Integrate with a Remote Provider

Q. What does a JMS client do to communicate with a remote JMS provider?

A. A typical client using the classic API performs these steps:

1. Look up a connection factory and destination using the configured naming service.
2. Create a Connection from the connection factory.
3. Create a Session, then create the required producer or consumer from that session.
4. For message consumption, start the connection when the application is ready to receive deliveries.
5. Close the application's JMS resources and naming context when finished.

The simplified API uses JMSContext in place of the separate Connection and Session. Follow the provider's supported API and the applicable standalone or container-managed lifecycle rules.

Q. What information do I need to set up communications with a remote JMS provider?

A. You will need the following information to set up communications with a remote JMS provider:

- The destination type: Whether the remote JMS destination is a queue or a topic.
- The JNDI name of the remote JMS destination.
- For durable subscriptions, the subscription name and, where required, the client ID. For MDBs, follow the documented MDB subscription and identity configuration; do not infer all identity values solely from the EJB name.
- For non-WebLogic remote JMS providers
  - Initial Context Factory Class Name: The Java class name of the remote JMS Provider's JNDI lookup service.
  - The file location of the Java JARs containing the remote JMS provider's JMS client and JNDI client libraries. Ensure that these JARs are specified in the local JVM's classpath.

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

- The URL of the remote provider's JNDI service. For WebLogic servers, the URL is usually in the form `t3://hostaddress:port`. If you are tunneling over HTTP, begin the URL with `http` rather than `t3`. No URL is required for server application code that accesses a WebLogic JMS Server that resides on the same WebLogic Server or WebLogic cluster as the application.
- The JNDI name of the remote provider's JMS connection factory. This connection factory must exist on the remote provider, not the local provider.

  For participation in a global JTA transaction, use a connection factory and integration configured for XA/global transactions. A local transacted session does not by itself require an XA-enabled factory.

  When the legacy default connection factories are enabled, familiar built-in JNDI names include the following:

  – `weblogic.jms.ConnectionFactory`: A non-XA capable factory.

  – `weblogic.jms.XAConnectionFactory`: An XA-capable factory

  – `weblogic.jms.MessageDrivenBeanConnectionFactory`: An XA-capable factory for message-driven EJBs.

  Applications can also use configured connection factories. Jakarta EE components have the platform default connection factory at java:comp/DefaultJMSConnectionFactory. Select the factory and transaction settings required by the application; do not treat this list as the complete factory inventory.

Q. What if a foreign JMS provider JNDI service has limited functionality?

A. The preferred method for locating JMS provider connection factories and destinations is to use a standard Jakarta EE JNDI lookup. Occasionally a non-WebLogic JMS provider's JNDI service is hard to use or unreliable. The solution is to create a startup class or load-on-start servlet that runs on a WebLogic server that does the following:

- Uses the foreign provider's proprietary (non-JNDI) APIs to locate connection factories and JMS destinations.
- Registers the JMS destinations and JMS connection factories in WebLogic JNDI.

Q. How can I pool JMS resources?

A. WebLogic-managed JMS connection-factory references can pool connections, sessions and producers. Applications should close their logical handles so eligible resources can be reused. Ordinary resource-reference consumers are closed with their provider consumers; do not assume they are pooled. MDB containers manage their own connections, sessions and consumers for message delivery.

Q. Which tools are available for integrating with remote JMS providers?

A. The following table summarizes the tools available for integrating with remote JMS providers:

Method	Automatic Enlistment	JMS Resource Pooling
Direct provider client	Depends on provider transaction integration and configuration; direct lookup alone adds none	No WebLogic-managed pool from direct use
Messaging bridge	Depends on configured QoS and adapter/provider transaction support	Managed internally by the bridge

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

Method	Automatic Enlistment	JMS Resource Pooling
Foreign JMS server definition	No transaction wrapper by itself; combine with an appropriate managed integration	No pool by itself
JMS resource reference	With the required managed transaction configuration and provider capability	Connections, sessions and producers; not ordinary consumers
JMS MDB	With supported container-managed transaction and provider configuration	Container-managed delivery resources
JMS SAF client	Local transacted sends are supported; do not assume participation in an application's global transaction	Not an application resource-reference pool
Server-side JMS SAF	A send to the local imported destination can participate when configured; later forwarding is asynchronous	Managed internally by SAF

Best Practices When Integrating with Remote Providers

Q. How do I receive messages from a remote JMS provider from within an EJB or Servlet?

A. Use a message driven EJB. Synchronous receives are not recommended because they idle a server side thread while the receiver blocks waiting for a message. See Using Message-Driven Beans.

Q. How do I send messages to a remote JMS provider from within an EJB or Servlet?

A. Use a resource reference. It provides pooling and automatic enlistment. See Using EJB/Servlet JMS Resource References. In limited cases where wrappers are not sufficient, you can write your own pooling code.

If the target destination is remote, then consider adding a local destination and messaging bridge to implement a store-and-forward high availability design. See Using a Messaging Bridge.

Another best practice is to use foreign JMS server definitions. Foreign JMS server definitions allow an application's JMS resources to be administratively changed and avoid the problem of hard coding URLs into application code. In addition, foreign JMS server definitions are required to enable resource references to reference remote JMS providers. See Using Foreign JMS Server Definitions.

Q. How do I communicate with remote JMS providers from a client?

A. If a standalone client needs a global transaction involving a non-WebLogic JMS provider, use a supported transaction integration. One option is an EJB service that performs the messaging work on WebLogic Server using appropriately configured managed resources and an XA-capable provider.

For asynchronous forwarding to a third-party JMS provider, consider a local destination and a messaging bridge. For supported WebLogic-to-WebLogic forwarding, consider server-side JMS SAF or the JMS SAF client, according to where local storage is needed. See Using a Messaging Bridge, Using WebLogic Store-and-Forward and Using WebLogic JMS SAF Client.

A client can also use the remote provider's naming and messaging APIs directly, or look up configured foreign mappings through WebLogic JNDI. Include the required compatible provider

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

libraries on the client classpath. Foreign mappings centralize lookup configuration; they do not themselves supply pooling or global transaction enlistment.

Q. How can I tune WebLogic JMS interoperability features?

A. See Tuning WebLogic Server EJBs, Tuning WebLogic Message Bridge, and Tuning WebLogic JMS Store-and-Forward in *Tuning Performance of Oracle WebLogic Server*.

Using Foreign JMS Server Definitions

Q. What are Foreign JMS Server Definitions?

A. Foreign JMS server definitions are an administratively configured symbolic link between a JNDI object in a remote JNDI directory, such as a JMS connection factory or destination object, and a JNDI name in the JNDI name space for a stand-alone WebLogic Server or a WebLogic cluster. They can be configured using the standard JMX MBean APIs, or programmatically using scripting. See Simplified Access to Foreign JMS Providers.

Q. When is it best to use a Foreign JMS Server Definition?

A. For this release, a Foreign JMS Server definition conveniently moves JMS JNDI parameters into one central place. You can share one definition between EJBs, servlets, and messaging bridges. You can change a definition without recompiling or changing deployment descriptors. They are especially useful for:

- Any message driven EJB (MDB) where it is desirable to administer standard JMS communication properties via configuration rather than hard code them into the application's EJB deployment descriptors. This applies even if the MDB's source destination isn't remote.
- Any MDB that has a destination remote to the cluster. This simplifies deployment descriptor configuration and enhances administrative control.
- Any EJB or servlet that sends or receives from a remote destination.
- Enabling resource references to refer to remote JMS providers. See Using EJB/Servlet JMS Resource References.

Using EJB/Servlet JMS Resource References

Q. What are JMS resource references?

A. A resource reference gives a component a logical name in its java:comp/env environment. Declare references through the supported annotations or component deployment descriptors and map them to the intended resources. For an application using the same compatible API, the deployment mapping can change without recompiling its resource lookup code.

WebLogic-managed connection-factory references can reuse eligible connection, session and producer resources after application handles are closed. They can also integrate with JTA when the provider and managed configuration support it.

Look up java:comp/env/<reference-name>, using the name actually declared for that component. For example, a reference named jms/QCF is looked up as java:comp/env/jms/QCF. The jms/ portion is part of the declared name, not an extra prefix added by the runtime.

Resource references provide no functionality outside of application code, and therefore are not useful for configuring a message driven EJB's source destination or a messaging bridge's source or target destinations.

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

For WebLogic documentation on JMS resource-reference pooling, see Enhanced Support for Using WebLogic JMS with EJBs and Servlets.

Q. What advantages do JMS resource references provide?

A. JMS resource references provide the following advantages:

- They ensure portability of servlet and EJB applications: they can be used to change an application's JMS resource without recompiling the application's source code.
- They provide automatic pooling of JMS Connection, Session, and MessageProducer objects.
- They provide automatic transaction enlistment for non-WebLogic JMS providers. This requires XA support in the JMS provider. If resource references are not used, then enlisting a non-WebLogic JMS provider with the current transaction requires extra programmatic steps.

Q. How do I use resource references with foreign JMS providers?

A. To enable resource references to reference remote JMS providers, they must be used in conjunction with a foreign JMS definition. This is because resources references do not provide a place to specify a URL or initial context factory. See Using Foreign JMS Server Definitions.

Q. How do I use resource references with non-transactional messaging?

A. Choose connection-factory transaction capabilities according to the application's transaction requirements. For XA-capable sessions used outside an existing JTA transaction, the WebLogic wrappers can create an internal transaction for an operation. The behavior depends on the provider and operation; WebLogic JMS producer operations with native transaction integration do not require that wrapper-created transaction. Do not assume that every operation through an XA-capable factory incurs the same overhead. See Understanding Transactions.

Using WebLogic Store-and-Forward

Q. What is the WebLogic Store-and-Forward Service?

A. The WebLogic Store-and-Forward (SAF) Service enables WebLogic Server to deliver messages reliably between applications that are distributed across WebLogic Server instances. For example, with the SAF service, an application that runs on or connects to a local WebLogic Server instance can reliably send messages to a destination that resides on a remote server. If the destination is not available at the moment the messages are sent, either because of network problems or system failures, then the messages are saved on a local server instance, and are forwarded to the remote destination when it becomes available. See Understanding the Store-and-Forward Service in *Administering the Store-and-Forward Service for Oracle WebLogic Server*.

Q. When should I use the WebLogic Store-and-Forward Service?

A. Use JMS SAF for asynchronous forwarding to WebLogic JMS destinations in supported release combinations. Choose the local sending service and imported destinations for the required topology.

- Between two stand-alone server instances
- Between server instances in a cluster
- Across two clusters in a domain
- Across separate domains

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

Q. What limitations and design tradeoffs should I consider for JMS SAF?

A. JMS SAF forwards sends through local imported destinations to supported WebLogic JMS destinations. It is not a general consumer for a remote source and does not forward directly to third-party JMS products; use an MDB, client consumer or messaging bridge as appropriate. Temporary destinations are not supported as SAF reply destinations. Confirm the supported source/target release combination in WebLogic Server Compatibility.

For direct access to an available local destination, send directly unless the topology requires forwarding. SAF adds a forwarding stage and local resource requirements; assess latency, throughput, storage and quotas for the workload. These are design tradeoffs, not a blanket prohibition based on latency.

Using WebLogic JMS SAF Client

Q. What is the WebLogic JMS SAF Client?

A. The JMS SAF client provides local store-and-forward for standalone producers sending to WebLogic JMS destinations. It accepts messages into the client's configured local SAF environment and forwards them when the target becomes available, subject to persistence, expiration, error-handling and resource-limit settings. See Reliably Sending Messages Using the JMS SAF Client.

Q. When should I use the WebLogic JMS SAF Client?

A. Use the JMS SAF client when a standalone producer needs local buffering during connectivity interruptions and the target WebLogic release combination is supported. It is a sending facility, not a general remote-consumer API. Consult the client guide's requirements and limitations.

Q. What are the limitations of using the JMS SAF Client?

A. See Limitations of Using the JMS SAF Client.

Using a Messaging Bridge

Q. What is a Messaging bridge?

A. Messaging bridges are administratively configured services that run on a WebLogic server. They automatically forward messages from a configured source JMS destination to a configured target JMS destination. These destinations can be on different servers than the bridge and can even be foreign (non-WebLogic) destinations. Each bridge destination is configured using the four common properties of a remote provider:

- The initial context factory
- The connection URL
- The connection factory JNDI name
- The destination JNDI name

Exactly-once bridge forwarding requires the corresponding QoS configuration and supported transaction-capable adapters, connection factories and providers. Do not infer that guarantee merely from a provider advertising XA capability. Configure whether QoS degradation is permitted and verify the bridge's effective QoS.

Q. When should I use a messaging bridge?

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

A. Use a bridge when the application needs asynchronous forwarding between supported JMS providers or destinations. A local source destination can let producers continue sending during a target outage, subject to source availability, quotas and expiration. To retain messages across failures, configure the required persistence and recovery; for a topic source, configure the required durable subscription. The bridge resumes forwarding when its connections and target become usable.

Q. When should I avoid using a messaging bridge?

A. Other methods are preferred in the following situations:

- Receiving from a remote destination :Use a message driven EJB or implement a client consumer directly.
- Sending messages to a local destination : Send directly to the local destination.
- Environment with low tolerance for message latency. Messaging Bridges increase latency and may lower throughput. Messaging bridges increase latency for messages as they introduce an extra destination in the message path and may lower throughput because they forward messages using a single thread.
- For supported WebLogic-to-WebLogic store-and-forward topologies, consider JMS SAF. See Using WebLogic Store-and-Forward.

Q. Why are some of my messages not being forwarded?

A. Usually, a messaging bridge should forward all messages. If some messages are not being forwarded, here are some possible reasons:

- Some messages may have an expiration time, in which case either the JMS provider for the source or target destination expires the message.
- If you configured the bridge source destination to specify a selector filter, then only the filtered messages are forwarded.
- A bridge does not directly provide an option to automatically move messages to an error destination or to automatically delete messages after a limited number of forward attempts. That said, it is possible that a JMS provider may provide such an option, which could affect messages on the bridge source destination. If a redelivery limit option is enabled on the JMS provider that hosts the bridge source destination, then you may need to reconfigure the provider to prevent the bridge automatic retry mechanism from causing messages to exceed the redelivery limit.

Using Message-Driven Beans

Q. What is a Message Driven EJB (MDB)?

A. A JMS message-driven bean (MDB) is an enterprise bean whose container delivers messages to its onMessage method. The container manages delivery resources and lifecycle according to the MDB configuration. MDBs have the following characteristics:

- Automatically connects to a source destination and automatically retries connecting if the remote destination is inaccessible.
- Can process received messages in container-managed transactions when the transaction settings and provider integration support it.
- Automatically pool their internal JMS connections, sessions, and consumers.
- Use supported annotations and deployment descriptors to configure delivery and transaction behavior; WebLogic settings and foreign mappings provide additional provider configuration.

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

- The messaging processing application logic is contained in a single method callback `onMessage()`.
- A MDB is an EJB that supports transactions, security, JDBC, and other typical EJB actions.

See Message-Driven EJBs in *Developing Jakarta Enterprise Beans Using Deployment Descriptors*.

Q. When should I use a MDB?

A. MDBs are the preferred mechanism for WebLogic Server applications that receive and process JMS messages.

Q. Do I need to use a Messaging Bridge with a MDB?

A. Configure MDBs to directly consume from their source destination rather than insert a messaging bridge between them. MDBs automatically retry connecting to their source destination if the source destination is inaccessible, so there is no need to insert a messaging bridge in the message path to provide higher availability. Introducing a messaging bridge may have a performance effect. See Using a Messaging Bridge.

Q. What is the best way to configure a MDB?

A. The following section provides tips for configuring a MDB:

- To configure MDB concurrency and thread pools, use the `max-beans-in-free-pool` and `dispatch-policy` descriptor fields. WebLogic Server may create fewer concurrent instances than `max-beans-in-free-pool` depending on the number of available server threads in the MDB's thread pool.
- Use foreign JMS server definitions when configuring a MDB to consume from a remote JMS provider. Although WebLogic MDB descriptors can be configured to directly refer to remote destinations, this information is packaged with the application and is not dynamically editable. You should configure a foreign JMS server definition and then configure the MDB to reference the foreign definition instead. Please note that some documentation refers to foreign JMS server definitions as wrappers. See Using Foreign JMS Server Definitions.
- For container-managed transactional delivery, configure Container transaction management and the Required transaction attribute for the listener method, using supported annotations or descriptors. The connection factory and provider integration must support the required transactions; third-party transactional JMS integration requires the appropriate XA capability. An incompatible configuration can fail deployment or connection setup rather than silently become non-transactional. The container establishes the transaction for message processing and enlists the supported resources.

Using AQ JMS

Q. Can I interoperate with AQ JMS?

A. Oracle WebLogic Server applications interoperate with Oracle Streams Advanced Queuing (AQ) through the JMS API using either WebLogic Server resources (Web Apps, EJBs, MDBs) or standalone clients. AQ JMS uses a database connection and stores JMS messages in a database accessible to an entire WebLogic Server cluster, enabling the use of database features and tooling for data manipulation and backup.

Use the JMS Foreign Server configuration to interoperate with Oracle Streams Advanced Queuing (AQ) through the JMS API using either WebLogic Server resources (Web Apps, EJBs, MDBs) or stand-alone clients. See Interoperating with Oracle AQ JMS in *Administering JMS Resources for Oracle WebLogic Server*.

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

B

How to Look Up a Destination

Learn how to use JNDI and a Create Destination Identifier to look up a message destination.

> **Note**
>
> For information about how to configure JMS resources, see Understanding JMS Resource Configuration in *Administering JMS Resources for Oracle WebLogic Server*.

Use a JNDI Name

For configured destinations, JNDI lookup is the recommended approach. Given an established javax.naming.Context named context and a String named destinationJndiName containing the configured JNDI name, use the appropriate lookup statement:

```
jakarta.jms.Queue queue = (jakarta.jms.Queue)
context.lookup(destinationJndiName);
```

```
jakarta.jms.Topic topic = (jakarta.jms.Topic)
context.lookup(destinationJndiName);
```

The destinationJndiName value is the destination's configured JNDI name. The binding must be visible through the selected naming context. See Using a JNDI Name and Examples of Syntax Used to Look Up Destinations.

Use a Create Destination Identifier

A Create Destination Identifier (CDI) is a WebLogic-specific destination name supplied to Session.createQueue() or Session.createTopic(), instead of looking up the destination by its application-visible JNDI name. The corresponding domain-specific QueueSession and TopicSession methods are also available. The classic Session methods have these signatures:

```
public Queue createQueue(
 String queueName
) throws JMSException
```

```
public Topic createTopic(
 String topicName
) throws JMSException
```

The syntax of the `queueName` and `topicName` strings is not defined by the JMS specification. For WebLogic JMS, the syntax is described here:

- Default WebLogic CDI Syntax
- Custom WebLogic CDI Syntax

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

> **Note**
>
> In WebLogic JMS, these methods return references to existing destinations; they do not provision new physical queues or topics. For information about creating destinations dynamically, see Using JMS Module Helper to Manage Applications.

Default WebLogic CDI Syntax

CDI syntax depends on whether the name identifies a physical destination or a logical distributed destination. For a standalone destination in a JMS system module, the default is JMS_Server_Name/JMS_Module_Name!Destination_Name. For a logical distributed destination, use its module-qualified configuration name. See Examples of Syntax Used to Look Up Destinations.

Custom WebLogic CDI Syntax

JMSCreateDestinationIdentifier supplies a custom CDI lookup name for a destination or distributed-destination member. It replaces the default lookup identifier for that physical destination; it is not the destination's JNDI name. See Examples of Syntax Used to Look Up Destinations.

The effective CDI lookup name must be unique among destinations on the same JMS server instance, including destinations from different modules. Destinations on different JMS server instances can use the same custom identifier.

> **Note**
>
> Check custom identifiers against both custom and default lookup names already used on the target JMS server. A conflicting name prevents successful destination deployment.

Server Affinity When Looking Up Destinations

Use ./ followed by the physical destination's CDI lookup identifier to restrict lookup to the WebLogic Server hosting the JMS connection. For example, use ./JMS_Module_Name!Destination_Name for a standalone system-module destination with its default identifier, or ./CDI_Name for a custom identifier. This locality is distinct from the host serving the application's JNDI context and from the connection factory's Server Affinity Enabled setting.

The lookup returns a matching started and available destination of the requested queue or topic type on that host. If more than one local JMS server has a match, no particular matching destination should be assumed. If none matches, lookup fails; it does not search other WebLogic Servers. The identifier after ./ must still identify the intended destination.

Examples of Syntax Used to Look Up Destinations

The following sections provide examples of the syntax used to reference a destination or a member of a distributed destination:

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

Non-Distributed Destinations

The following section provides examples of syntax used to reference regular destinations (destinations that are not distributed):

JNDI Syntax for Non-Distributed Destinations

Most applications use JNDI instead of CDI to look up destinations. The following section provides examples of the syntax used to reference non-distributed destinations using JNDI:

- When a JNDI name is configured, a string defined by:

  ```
  Dest_JNDI_Name
  ```

- When a local JNDI name is configured:

  ```
  Dest_Local_JNDI_Name
  ```

> **Note**
>
> The local JNDI name only works when the JNDI context host is on the same server as the non distributed destinations. The JNDI context host is not necessarily the same as the JMS connection host.

CDI Syntax for Non-Distributed Destinations

This section shows CDI forms for a non-distributed destination using createQueue() or createTopic():

- When using the default CDI, a string defined by:

  ```
  JMS_Server_Name/JMS_Module_Name!Destination_Name
  ```

- When a custom CDI is configured, a string defined by:

  ```
  JMS_Server_Name/CDI_Name
  ```

> **Note**
>
> Replacing JMS_Server_Name with . restricts CDI lookup to the JMS connection host; the remaining lookup identifier is unchanged.

Uniform Distributed Destinations

The following section provides examples of the syntax used to reference Uniform Distributed Destinations (UDDs):

JNDI Syntax for UDDs

Most applications use JNDI instead of CDI to look up destinations. The following section provides examples of how to reference an individual member or logical UDD using JNDI:

- For a logical UDD, a string defined by:

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

```
udd-jndi-name
```

- For an individual member of a UDD hosted on a set of individually configured JMS servers, a string defined by:

  ```
  jms-server-name@udd-jndi-name
  ```

- For an individual member of a UDD hosted on a cluster targeted JMS server, a string defined by:

  ```
  jms-server-name@wl-server-name@udd-jndi-name
  ```

  Here, wl-server-name is the WebLogic Server name used to identify the JMS server instance. For a dynamic server, use its configured prefix and server number, respecting the configured starting index. The instance identity is distinct from its current hosting server after migration.

CDI Syntax for UDDs

> **Note**
>
> JMSModuleHelper.`uddMemberName`(jmsServerName, uddName) constructs the member configuration name. JMSModuleHelper.`uddMemberJNDIName`(jmsServerName, uddJndiName) constructs the member JNDI name. Neither returns a complete CDI argument. For CDI, use the appropriate JMS server instance and module-qualified member name shown below.

The following forms apply to UDDs in JMS system modules. Let jms-server-instance-name be the name of the JMS server instance for the member: the JMS server name for an individually targeted server, or jms-server-name@wl-server-name for an instance of a cluster-targeted distributed JMS server. Use the actual instance identity; do not substitute a different current host name merely because the service migrates.

- Physical member with its default CDI identifier: jms-server-instance-name/module-name! jms-server-instance-name@udd-name.
- Physical member with a custom CDI identifier: jms-server-instance-name/cdi-name, where cdi-name is the effective custom identifier for that member.
- Logical UDD: module-name!udd-name.

For example, for JMS server instance OrdersJMS@Managed1, system module OrdersModule, and UDD OrdersQueue, the default physical-member CDI argument is OrdersJMS@Managed1/OrdersModule!OrdersJMS@Managed1@OrdersQueue. The logical argument is OrdersModule!OrdersQueue. These are configuration names, not JNDI names.

> **Note**
>
> For a physical UDD member, ./ restricts lookup to the JMS connection host. The remaining identifier must match that member's effective CDI name; with a default identifier, this includes the member-name prefix. This applies to both queues and topics. It does not request an arbitrary local member of the logical UDD. Lookup fails if there is no eligible local match.

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

Weighted Distributed Destinations

> **Note**
>
> Weighted distributed destinations have been deprecated since WebLogic Server 10.3.4.0. Oracle recommends uniform distributed destinations.

A weighted distributed destination (WDD) groups individually configured physical destinations. Look up the logical WDD by its configured JNDI name. The logical WDD and each physical member are different destination objects; looking up a member does not return the logical WDD.

JNDI Syntax for WDDs

The following section provides examples how to reference an individual member or logical WDD using JNDI:

- For a logical WDD, a string defined by:

  ```
  wdd-jndi-name
  ```

- For an individual WDD member, use that physical destination's JNDI name. See JNDI Syntax for Non-Distributed Destinations.

CDI Syntax for WDDs

For a logical WDD in a JMS system module, supply module-name!wdd-name to createQueue() or createTopic(), using the WDD's configuration name and matching destination type. The logical destination must be deployed and known to the server handling the lookup. A physical member's JNDI name is not the logical WDD's CDI argument.

For an individual member, use that physical destination's default or custom CDI form. The ./ prefix can restrict physical-member lookup to the JMS connection host. See CDI Syntax for Non-Distributed Destinations.

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

C

Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API

Learn how to design a distributed application or a container that offers high availability (HA), scalability, and flexibility when using JMS distributed destinations in a clustered environment.

> **Note**
>
> Oracle recommends using message-driven beans (MDBs) where appropriate for applications that consume from distributed destinations. The MDB container manages consumers and recovery according to the destination type and configured distribution strategy, together with security, threading, pooling, lifecycle, and transaction integration. Applications that implement their own consumer management can use the Destination Availability Helper to respond to membership changes. The helper provides notifications; the application remains responsible for consumer creation, cleanup, and recovery.

Introduction

A distributed destination groups physical queues or topics behind a logical destination. Producer member selection and message distribution depend on whether the destination is a distributed queue, a partitioned distributed topic, or a replicated distributed topic. Consumer applications must provide the member coverage required by their delivery strategy and handle failures.

Distributed destination users that don't leverage MDBs may encounter problems with consumer applications. These include:

- Failing to ensure that all DD members are serviced by consumers.
- Unprocessed messages accumulating on DD members that have no consumers.
- DD Consumers not automatically rebalancing in the event of a JMS server migration, WebLogic Server restart, or any other event that results in DD member changes.

To address these use cases, WebLogic Server provides the JMS Destination Availability Helper APIs and advanced topic features in Developing Advanced Pub/Sub Applications.

Controlling DD Producer Load Balancing

Before discussing consumer load balancing, it is helpful to first explore producer load balancing basics and best practices.

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

Basic JMS

A classic JMS application prepares to send messages as follows:

1. Look up a connection factory and the destination, and create a Connection from the factory.
2. Create a Session from the connection, then create a MessageProducer from the session.
3. Send messages using the producer and the Destination object, either supplied when the producer is created or to the applicable send method.

The WebLogic Server hosting the JMS connection is the connection host. Connection-host selection depends on the connection factory's deployment, configuration, and server availability. The connection host routes operations to the destination's hosting service; it is not necessarily the destination host.

To use provider member selection for a distributed queue or partitioned distributed topic, look up the logical destination's JNDI name and use the returned Destination object. Use a physical member destination only when the application deliberately manages member selection.

Senders to Distributed Queues (DQs) and Partitioned Distributed Topics (PDTs)

When a producer sends to a logical distributed queue or partitioned distributed topic with Load Balancing Enabled, WebLogic selects eligible members using the configured load-balancing algorithm and applicable affinity and messaging rules. Server Affinity Enabled can favor eligible members on the connection host; it does not by itself define the complete selection order.

To allow selection without server affinity, use a custom connection factory with Server Affinity Enabled set to false and Load Balancing Enabled set to true. This does not guarantee round-robin selection across every active member. The destination's load-balancing algorithm, member availability, transactions, and Unit-of-Order requirements can affect selection.

Senders to Replicated Distributed Topics (RDTs)

A producer sending to a logical replicated distributed topic normally selects a member and continues using it. Unit-of-Order routing and failure handling can affect that selection. The RDT forwards published messages to its other members; delivery to subscriptions remains subject to normal JMS delivery and message-selection rules.

Server Affinity Enabled can influence the initial member selection. Set it to false on the connection factory if the application does not want that preference. Load Balancing Enabled does not cause an RDT producer to select a different member on each send.

Using the JMS Destination Availability Helper API

The following sections provide information on how to use the `JMSDestinationAvailabilityHelper` APIs:

Overview

For consumer APIs that support a logical distributed destination, creating a consumer selects a member. Existing consumers do not automatically redistribute themselves when new members

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

become available. APIs that require a physical topic must instead use a member destination; see Considerations When Using Standard Shared Subscriptions.

The helper discovers destinations and reports membership changes for supported WebLogic distributed destinations. Register a listener using the destination's logical JNDI name to discover its members, including members that become available after application startup. A regular destination or third-party JMS destination provides only an initial successful-lookup notification, not continuing availability monitoring.

The application chooses which notified members need consumers for its delivery strategy. MDBs use destination discovery as part of their managed consumer lifecycle; applications using the helper directly must implement their own lifecycle and recovery.

General Flow

Applications that use the `JMSDestinationAvailabilityHelper` APIs should follow these general steps:

1. Implement weblogic.jms.extensions.DestinationAvailabilityListener to handle the callbacks described in step 3.
2. Register the destination's JNDI name and a listener with JMSDestinationAvailabilityHelper. The following complete helper method accepts an application-supplied JNDI environment and listener and returns the RegistrationHandle. An in-server application using its local naming context can supply null for the environment. For a remote naming service, supply the provider URL, initial context factory, and appropriate security configuration. Retain the handle until the registration is no longer needed.

```
import java.util.Hashtable;
import weblogic.jms.extensions.DestinationAvailabilityListener;
import weblogic.jms.extensions.JMSDestinationAvailabilityHelper;
import weblogic.jms.extensions.RegistrationHandle;

public final class DestinationRegistration {
    public static RegistrationHandle register(
            Hashtable<String, Object> environment,
            String destinationJndiName,
            DestinationAvailabilityListener listener) {
        Hashtable<String, Object> properties = environment == null
                ? null : new Hashtable<>(environment);
        return JMSDestinationAvailabilityHelper.getInstance()
                .register(properties, destinationJndiName, listener);
    }
}
```

3. Handle the callbacks and coordinate shared state with application worker and recovery tasks. Do not assume that reusing one listener object for multiple registrations serializes its callbacks; use a separate listener per registration or appropriate synchronization.
   - onDestinationsAvailable(): The list contains newly available or newly rediscovered members, not a complete snapshot on every callback. Update the application's member state and arrange consumer creation for the required members. Availability can change before creation succeeds.
   - onDestinationsUnavailable(): For a logical distributed-destination registration, remove the reported members from application state, cancel their pending consumer-creation work, and close affected consumers. Other members remain in the application state.
   - onFailure(): Report discovery failures without repeatedly flooding the log. The helper retries discovery internally, but configuration or security errors can require administrator action. Helper retries do not recreate the application's JMS consumers.

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

4. At application shutdown or undeployment, call unregister() on the RegistrationHandle, cancel application retry work, and close the JMS resources and JNDI contexts owned by the application. Unregistering the listener does not close application-created JMS resources.

Handling the weblogic.jms.extensions.DestinationDetail

Each callback entry implements weblogic.jms.extensions.DestinationDetail. Use getType() to identify its destination type and getJNDIName() to obtain the JNDI name represented by the entry.

For a member of a WebLogic logical distributed destination, getType() returns DESTINATION_TYPE_DD_QUEUE, DESTINATION_TYPE_REPLICATED_DT, or DESTINATION_TYPE_PARTITIONED_DT, and getJNDIName() identifies the member. Regular destinations use DESTINATION_TYPE_PHYSICAL_QUEUE or DESTINATION_TYPE_PHYSICAL_TOPIC. Third-party destinations use DESTINATION_TYPE_FOREIGN_QUEUE or DESTINATION_TYPE_FOREIGN_TOPIC. These constants are declared by DestinationDetail.

For distributed members, isLocalWLSServer() identifies membership on the WebLogic Server running the caller; isLocalCluster() identifies membership in that server's cluster. Cluster-local does not mean the same JVM. These flags do not establish local placement for regular or third-party destinations. Choose member coverage according to the application strategy before applying a locality optimization. See Best Practice for Local Server Consumers.

For the complete field and method contracts, see DestinationDetail in *Java API Reference for Oracle WebLogic Server*.

Best Practices for Consumer Containers

The following sections provide best practice guidelines for consumer containers:

When to Register and Unregister

1. Register when the application starts or is deployed. Handle immediate registration errors as well as onFailure() callbacks. Temporary discovery failures need not prevent application startup, but expose configuration and security errors so that they can be corrected.
2. Unregister at shutdown or undeployment, cancel application retry tasks, and close resources owned by the application.

URL Handling

1. For an application running in WebLogic Server and using that server's local naming context, omit the provider URL. A standalone client or an application accessing a remote naming service must supply the appropriate provider URL and initial context factory.
2. Use DestinationDetail.isLocalWLSServer() and isLocalCluster() to evaluate distributed-member locality. Do not infer that a remote naming URL is incorrect solely because a returned member is local; context selection also depends on the intended naming and security environment.

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

Failure Handling

1. Log discovery failures and avoid repeating identical messages excessively. Distinguish temporary unavailability from an invalid JNDI name, provider URL, or security configuration. The helper retries discovery; administrator action may still be required.
2. When the JMS connection fails, arrange cleanup and application-managed recovery outside the callback. Recreate the failed resources and restore the consumers required by the current member set. For other JMS exceptions, determine whether the operation can be retried or requires a configuration or application correction; do not treat every exception as a failed connection.
3. Use delayed retries for transient failures and cancel them when the member is removed or the application stops. Coordinate retries with availability callbacks so that stale work does not recreate unwanted consumers.

JNDI Context Handling

> **Note**
>
> Reuse a JNDI context only within a compatible provider and security scope, observing the provider's threading requirements. Use separate contexts when the application needs different domains or identities; see Understanding JMS Security.

Close contexts when they are no longer needed. For thread-based WebLogic security, create and close a context on the same thread and close nested contexts in reverse order so that the previous subject is restored. Passing a context to a different thread does not transfer the originating thread's subject.

Recreate a context when it is no longer usable. A failed lookup can instead indicate an incorrect name or unavailable binding; simply recreating the context does not correct every lookup failure.

JMS Connection Handling

1. For application-owned JMS connections, register an ExceptionListener to observe connection failures. Container-managed JMS resources must follow the applicable component lifecycle and restrictions.
2. Arrange recovery outside the exception callback using an execution mechanism allowed by the application's environment. Close failed application-owned resources and retry connection and consumer creation with a delay; coordinate this work with helper membership notifications.
3. Cancel recovery work and close application-owned connections at shutdown or undeployment. Do not close resources owned by the container.
4. Choose the number of connections and sessions according to concurrency, connection-host distribution, and resource usage. Additional connections may improve distribution but consume resources; measure the effect for the intended workload and provider.

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

Interoperability Guidelines

The JMSDestinationAvailabilityHelper in *Java API Reference for Oracle WebLogic Server* includes details about usage and behavior of the various methods available, including details about interoperability guidelines discussed in the following sections:

API Availability

JMSDestinationAvailabilityHelper is a WebLogic extension available to supported Java client and server environments that include this API. The legacy behavior described below explains compatibility handling; it does not establish that every older client or server version is supported with this release.

Foreign Contexts

The registration environment identifies the JNDI provider from which to resolve the destination. It can identify a supported WebLogic naming context or a compatible third-party context. Naming-provider availability alone does not establish JMS provider or release compatibility.

For foreign (non-WebLogic) contexts, the foreign JNDI vendor's classes must be in the current classpath and the `Context.INITIAL_CONTEXT_FACTORY` property must reference the foreign vendor JNDI context factory class name.

Destination Type Support

The helper can resolve WebLogic JMS and compatible third-party JMS destinations through JNDI. A Foreign JMS Server mapping is evaluated according to the destination to which it resolves: a WebLogic logical distributed destination can provide member notifications, whereas a third-party destination provides one initial availability notification. Do not use that initial notification as proof of continuing availability.

Unavailable Notifications

Unavailable notifications apply to member changes discovered through a WebLogic logical distributed-destination registration. Registrations for regular destinations, individual physical member destinations, and third-party destinations do not provide this monitoring. Such registrations provide one initial availability notification after successful resolution. Applications must still handle JMS operation failures.

Historical Behavior for Distributed Destinations Before WebLogic Server 9.0

The helper's compatibility handling distinguishes WebLogic distributed destinations before 9.0 from later distributed destinations. For a pre-9.0 destination, it reports one initial entry containing the logical JNDI name, with getType() returning DESTINATION_TYPE_PHYSICAL_QUEUE or DESTINATION_TYPE_PHYSICAL_TOPIC. It does not report individual member availability changes.

For later distributed destinations, a logical-destination registration can report individual members. These historical distinctions are not a statement of the client/server combinations supported by this release.

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

Historical Behavior for Distributed Topics Before WebLogic Server 10.3.4

The advanced WebLogic topic features used by the policy-based sharing patterns were introduced in WebLogic Server 10.3.4. Check DestinationDetail.isAdvancedTopicSupported() before applying those patterns to a discovered WebLogic topic. Do not infer support solely from a topic's JNDI name or treat third-party topic behavior as a WebLogic contract.

For historical replicated topics without these features, a non-durable consumer on one member can receive replicated publications, subject to normal delivery and selection rules; the all-member sharing patterns below must not be assumed to apply. Durable subscriptions require explicit physical-member selection and an appropriate identity and recovery design. Creating a durable consumer on a logical distributed-topic object is not supported.

> **Note**
>
> For the policy configuration, see Configure an Unrestricted ClientID and Configure Shared Subscriptions in *Administering JMS Resources for Oracle WebLogic Server*.

DestinationDetail Fields

The information available in a DestinationDetail depends on the destination type and provider. See DestinationDetail in *Java API Reference for Oracle WebLogic Server*.

Security Considerations

The following sections provide information about implementing security using the Jakarta EE and WebLogic Server security models:

WebLogic Server Security Model

With thread-based JMS security, WebLogic uses the current thread's subject for subsequent JNDI and JMS operations. Creating a JMS resource under a subject does not permanently associate that subject with every later thread-based operation. Object-based security (OBS) instead associates a subject with the relevant JMS objects. See Understanding JMS Security.

Passing Credentials Between Threads

Passing a JNDI context or JMS object to another thread does not, by itself, establish the originating thread's subject on that thread. For thread-based security, run the operations under the appropriate subject. OBS has different identity-association rules; see Understanding JMS Security.

Using the Same Thread

For thread-based security, create the naming context, perform the related operations under the intended subject, and close the context on the same thread. If contexts are nested, close them in reverse creation order. Use explicit subject selection when operations for different identities are interleaved.

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

Pass as Anonymous User

Use an anonymous subject only when the intended JNDI and JMS operations are authorized for anonymous users. Anonymous access is not a general solution for cross-domain security. Configure the appropriate identity and domain security for the operation; see Managing Cross-Domain Security.

Cache and Reuse a Subject from the Initial Context

The following helper captures the current subject when it is constructed. Construct it while the intended subject is active, then use run() to execute an operation under that subject on the calling thread, including an application worker thread. It does not create an anonymous subject or extract a subject from a context passed from another thread. PrivilegedActionException is propagated to the caller for handling.

```
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import javax.security.auth.Subject;
import weblogic.security.Security;

public final class CapturedSubject {
    private final Subject subject;

    public CapturedSubject() {
        subject = Security.getCurrentSubject();
    }

    public Object run(PrivilegedExceptionAction<?> action)
            throws PrivilegedActionException {
        return Security.runAs(subject, action);
    }
}
```

Managing Cross-Domain Security

When using the `JMSDestinationAvailabilityHelper` API in communication between different WebLogic domains, see Cross-Domain Security Guidelines.

Authentication of Users

Authentication establishes the user's identity. Authorization separately determines whether that identity can perform the requested JNDI and JMS operations. The following sections describe where credentials are supplied and which operations use the resulting subject.

Specifying Credentials for a JNDI Context

Supply the appropriate credentials when creating a naming context that requires authentication. For the thread-based WebLogic context, the resulting subject applies to subsequent operations on that thread. An in-server local context created without a provider URL or credentials uses the existing thread identity. Closing the context on its creating thread restores the preceding subject; close nested contexts in reverse order.

These thread-based rules do not describe OBS context behavior. See Understanding JMS Security for OBS defaults, explicit JMS credentials, and subject selection across domains.

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

Specifying Credentials for a JMS Connection

The username and password supplied to createConnection() or createContext() are used to authenticate the connection request. With thread-based WebLogic JMS security, subsequent operations use the current thread's subject; supplying connection credentials does not set that thread's identity for later operations. With OBS, explicit JMS credentials can select the subject associated with the JMS objects. See Understanding JMS Security.

Using Credentials of a Foreign JMS Server JNDI Context

Configure the Foreign JMS Server's naming environment for access to the remote JNDI provider. Specify the naming principal through the appropriate JNDI property and the naming credential through JNDI Properties Credential. These naming credentials are distinct from the username and password configured on a Foreign Connection Factory mapping.

Using Credentials of a Foreign JMS Server Connection

The username and password on a Foreign Connection Factory mapping are used by supported WebLogic integrations, including MDB connections and managed resource-reference access. A direct lookup of the underlying connection factory does not automatically supply these credentials to an application's createConnection() or createContext() call. See Enhanced Support for Using WebLogic JMS with EJBs and Servlets.

Securing Destinations

Use JMS resource security policies to authorize operations on destinations. See Java Messaging Service (JMS) Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Securing Wire Data

Use appropriately configured TLS connections when JMS data must be protected in transit. See Configuring SSL in *Administering Security for Oracle WebLogic Server*.

Transaction Considerations

JTA transaction association is thread-based. WebLogic JMS operations can enlist automatically in the current WebLogic JTA transaction when the connection factory and resource integration support that participation. Do not assume that the presence of a transaction alone makes every JMS resource transactional. See Automatically Enlisting Transactions.

For a third-party JMS provider, use a supported transaction integration, such as an appropriately configured managed resource reference with an XA-capable provider. A standalone application must not assume that WebLogic JMS automatically enlists with an arbitrary third-party transaction manager. See FAQs: Integrating Remote JMS Providers.

A native locally transacted JMS session has its own commit and rollback boundary; it is not the same as a container-managed session participating in JTA. An XA-capable factory does not prevent the applicable native API from creating a local transaction. Follow the programming model and transaction rules described in Using Transactions with WebLogic JMS.

Keep transaction work on the associated thread unless the application deliberately uses a supported transaction suspend/resume mechanism. Do not assume that handing a JMS resource to a worker thread transfers the transaction.

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

Strategies for Uniform Distributed Queue Consumers

A UDQ consumer application can run inside WebLogic Server or as a standalone client. Server-side applications may run with the destination services or access them remotely.

> **Note**
>
> Oracle recommends MDBs where appropriate so that the container manages consumer lifecycle and recovery.

The following guidance applies to applications that manage their own consumers.

General Strategies

Design the application's consumer population to cover the UDQ members that can hold messages for its workload. Register the logical UDQ JNDI name with JMSDestinationAvailabilityHelper. Each registration reports newly available members and subsequent removals; keep an application member set rather than treating every notification as a full snapshot.

One straightforward strategy is for each application instance to create consumers on every available member. Another is to distribute member coverage among instances. For each newly available member assigned to an instance, create the required consumer or consumers and arrange delayed recovery if creation fails. For each unavailable member, remove its state, cancel pending creation work, and close its consumers.

Consumers on a queue compete for eligible messages; member coverage does not reserve every message for this application when other consumers exist. If the design relies on configured queue forwarding, account for its routing, ordering, and availability behavior. See Best Practice for Local Server Consumers.

Best Practice for Local Server Consumers

Consuming from members on the application's WebLogic Server can reduce network traffic when the JMS connection host is also local. Use DestinationDetail.isLocalWLSServer() to identify those members. isLocalCluster() identifies the same cluster and does not establish same-server placement.

Restrict consumption to local members only when application instances collectively cover all required members, including after service migration or membership changes. Otherwise provide remote consumers or another appropriate coverage strategy. Local-only consumption can also leave processing capacity unevenly used when member backlogs differ; evaluate distribution and recovery for the workload.

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

Strategies for Subscribers on Uniform Distributed Topics

> **Note**
>
> Oracle recommends using MDBs to implement advanced message distribution modes using replicated and partitioned distributed topics. For detailed information about advanced publish/subscribe application design using MDBs, see Developing Advanced Pub/Sub Applications and Configuring and Deploying MDBs Using Distributed Topics in *Developing Message-Driven Beans for Oracle WebLogic Server*.

The following recipes describe WebLogic policy-based subscription sharing through traditional subscriber APIs. Configure Client ID Policy as Unrestricted and Subscription Sharing Policy as Sharable when using those recipes, and create subscribers on physical member destinations discovered through JMSDestinationAvailabilityHelper. These are not universal prerequisites for the standard named shared-subscription APIs. See Shared Subscriptions and Client ID Policy and Considerations When Using Standard Shared Subscriptions.

A replicated distributed topic (RDT) forwards publications among its members. A partitioned distributed topic (PDT) does not replicate each publication to the other members; subscribers must cover the appropriate members to receive the application's publications. Normal message selection, delivery, expiration, and durable-subscription retention rules still apply.

Choose the one-copy-per-instance or one-copy-per-application strategy according to the required application behavior. The following sections describe member coverage and subscription identity for each strategy.

See Configuring Partitioned Distributed Topics in *Administering JMS Resources for Oracle WebLogic Server*.

One Copy Per Instance

With the one-copy-per-instance strategy, each application instance has its own subscription and receives its own copy of messages published to the topic, subject to normal JMS delivery and message-selection rules. Use the following policy-based recipe when each instance must process the messages independently.

General Pattern Design Strategy for One Copy Per Instance

For the all-member policy-based strategy, each application instance performs the following steps.

1. Choose a client ID unique to the application instance, such as an application base ID plus a stable instance identifier. Use that same instance identity on its connections. For durable consumption, choose a stable subscription name and reuse it when reopening that instance's subscriptions.
2. Create the connections and sessions. Configure Client ID Policy as Unrestricted and Subscription Sharing Policy as Sharable before creating subscribers. Within an instance, consumers intended to share a physical subscription must use matching subscription details; different instances must not accidentally share the same subscription identity.

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

3. Register the logical distributed-topic JNDI name with JMSDestinationAvailabilityHelper and maintain the reported member set. Arrange application-owned connection failure handling and delayed recovery outside callbacks.
4. Create subscribers for this instance on every available member. On a PDT, this coverage is required because publications are partitioned among members. A matching subscription identity on another member still identifies a separate physical subscription.
5. For an RDT in this all-member strategy, exclude forwarded replicas. With no application selector, use NOT JMS_WL_DDForwarded. With an application selector, use (NOT JMS_WL_DDForwarded) AND (<application-selector>), replacing the placeholder with the complete selector expression. Preserve the parentheses around the entire application selector. Do not apply this filter to a strategy that depends on receiving replicated copies from one RDT member.
6. When a member becomes unavailable, cancel its pending creation work and close its consumers. Recreate the required consumers when it is reported available again. Preserve durable identity when reopening retained subscriptions; do not unsubscribe as routine failure cleanup.
7. At shutdown, unregister, cancel recovery tasks, and close application-owned resources. Manage durable-subscription deletion separately from consumer closure.

Best Practice for Local Server Consumers using One Copy Per Instance

For a PDT, each application instance that needs its own copy of all publications must cover every member, including remote members. Limiting that instance to members on its own server does not implement the one-copy-per-instance strategy. Follow General Pattern Design Strategy for One Copy Per Instance.

For non-durable consumption from an RDT, an alternative is to select one available member per application instance and receive that member's replicated publications. A local member may reduce network traffic. Do not apply NOT JMS_WL_DDForwarded in this one-member design, because it relies on forwarded copies. Handle member failure and selection of a replacement; a non-durable subscription does not retain messages while it is absent.

For durable consumption, retain the all-member strategy unless the application explicitly manages durable member identity, retained backlogs, and recovery. Creating a subscription on a different member does not transfer the old member's durable backlog.

Use DestinationDetail.isLocalWLSServer() when evaluating locality. DestinationDetail is the top-level weblogic.jms.extensions interface. Locality is an optimization only after the required delivery and recovery behavior has been established.

One Copy Per Application

With the one-copy-per-application strategy, application instances share subscriptions so that each publication is delivered to the application as a whole rather than separately to every instance, subject to normal JMS delivery and message-selection rules. Messages can be distributed among instances so that they share the processing work. Failures and redelivery can still result in repeated processing.

The following sections describe the policy-based design and its local-consumer optimization.

General Pattern Design Strategy for One Copy Per Application

For policy-based one-copy-per-application consumption, coordinate the application instances as follows.

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

1. Choose one application client ID to use on all participating connections. For durable consumption, choose one common application subscription name; do not derive a different name for each application instance. Reuse the same identity when reopening the application's subscriptions.
2. Configure Client ID Policy as Unrestricted and Subscription Sharing Policy as Sharable before creating subscribers. For policy-based non-durable sharing, match the physical topic, client ID and policy, selector, and noLocal setting. For durable sharing, use the common client ID and subscription name with compatible subscription settings.
3. Register the logical distributed-topic JNDI name with JMSDestinationAvailabilityHelper. Maintain member state and arrange application-owned connection failure handling and delayed recovery outside callbacks.
4. Ensure that the application instances collectively provide subscribers on every available member required by the strategy. Multiple instances may share the same physical member's subscription. Matching keys on different members identify separate physical subscriptions; a common client ID does not merge them across the distributed topic.
5. For an RDT, use NOT JMS_WL_DDForwarded to exclude replicated copies. If an application selector is also needed, use (NOT JMS_WL_DDForwarded) AND (<application-selector>), substituting the complete selector and retaining its parentheses. For a PDT, consume each member's publications without that RDT replica filter.
6. For each unavailable member, cancel pending creation work and close affected consumers. Restore required coverage when the member becomes available again. Preserve durable identity and retained subscriptions during ordinary recovery.
7. At shutdown, unregister and cancel retry work, then close application-owned resources. Coordinate durable-subscription deletion separately; closing a consumer does not delete its durable subscription.

Best Practice for Local Server Consumers Using One Copy Per Application

The application can limit each instance to local members when its instances collectively cover every required distributed-topic member, including after service migration or membership changes. Use DestinationDetail.isLocalWLSServer() to identify a member on the calling instance's WebLogic Server. Same-cluster placement alone does not provide same-server locality or complete coverage.

Retain the common application subscription identity and the other requirements in General Pattern Design Strategy for One Copy Per Application. For an RDT, retain the NOT JMS_WL_DDForwarded selector so that the application consumes the original publication at its originating member rather than additional replicated copies. For a PDT, collectively cover the members containing the partitioned publications.

If a member has no local application instance, provide a remote consumer or another recovery arrangement that restores the required coverage. Evaluate workload distribution as well as locality; local-only consumption is an optimization, not a replacement for member and failure handling.

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