Oracle® Fusion Middleware

Developing JMS Applications for Oracle WebLogic Server

14c (14.1.1.0.0)
F18322-04
January 2023

ORACLE

Oracle Fusion Middleware Developing JMS Applications for Oracle WebLogic Server, 14c (14.1.1.0.0)

F18322-04

Contents

5 Developing a Basic JMS Application

6 Managing Your Applications

8 Using Multicasting with WebLogic JMS

9 Using Distributed Destinations

10 Using the Message Unit-of-Order

11 Using Unit-of-Work Message Groups

13 Developing Advanced Pub/Sub Applications

14 Recovering from a Server Failure

A Server Session Pools (Deprecated)

B FAQs: Integrating Remote JMS Providers

C How to Look Up a Destination

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

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 Java 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.

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.

Avitek Medical Records Application (MedRec) and Tutorials

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

MedRec demonstrates WebLogic Server and Java EE features, and highlights Oracle-recommended best practices. MedRec is optionally installed with the WebLogic Server installation. You can start MedRec from the *ORACLE_HOME*\user_projects\domains\medrec directory, where *ORACLE_HOME* is the directory you specified as the Oracle Home when you installed Oracle WebLogic Server.

MedRec includes a service tier that is made up of Enterprise Java Beans (EJBs) that work together to process requests from web applications, web services, and workflow applications, and future client applications. The application includes message-driven, stateless session, stateful session, and entity EJBs.

New and Changed JMS Features in This Release

This release includes the following new and changed features for WebLogic Server 12.x:

- WebLogic Server 12.2.1 supports the use of simplified APIs specified by JMS 2.0. See Understanding the Simplified API Programming Model .

- Weighted Distributed Destinations are deprecated in WebLogic Server 10.3.4.0. Oracle recommends using Uniform Distributed Destinations.
- Advanced WebLogic JMS publish and subscribe (pub/sub) concepts and functionality of Uniform Distributed Topics (UDTs) necessary to design high availability applications. See Developing Advanced Pub/Sub Applications.
- The `JMSDestinationAvailabilityHelper` API provides a means for getting notifications when destinations become available or unavailable. These APIs are for advanced use cases only. Use this helper only when standard approaches for solving WebLogic distributed consumer problems have been exhausted. See Using the JMS Destination Availability Helper APIs with Distributed Queues in *Developing JMS Applications 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. See Security Considerations for Anonymous Users.

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

Conventions

The following text conventions are used in this document:

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

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.
It is assumed that you are familiar with Java programming and JMS 1.1 and JMS 2.0 concepts and features.

Overview of the Java Message Service and WebLogic JMS

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

WebLogic JMS fully supports the JMS Specification, described at `http://www.oracle.com/technetwork/java/jms/index.html`, and also provides numerous WebLogic JMS Extensions that go above and beyond the standard JMS APIs.

What Is the Java Message Service?

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, allowing you to separate the details nation system from the application code.

The Java Message Service (JMS) is a standard API for accessing enterprise messaging systems. Specifically, JMS:

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

Figure 2-1 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.

Implementation of Java Specifications

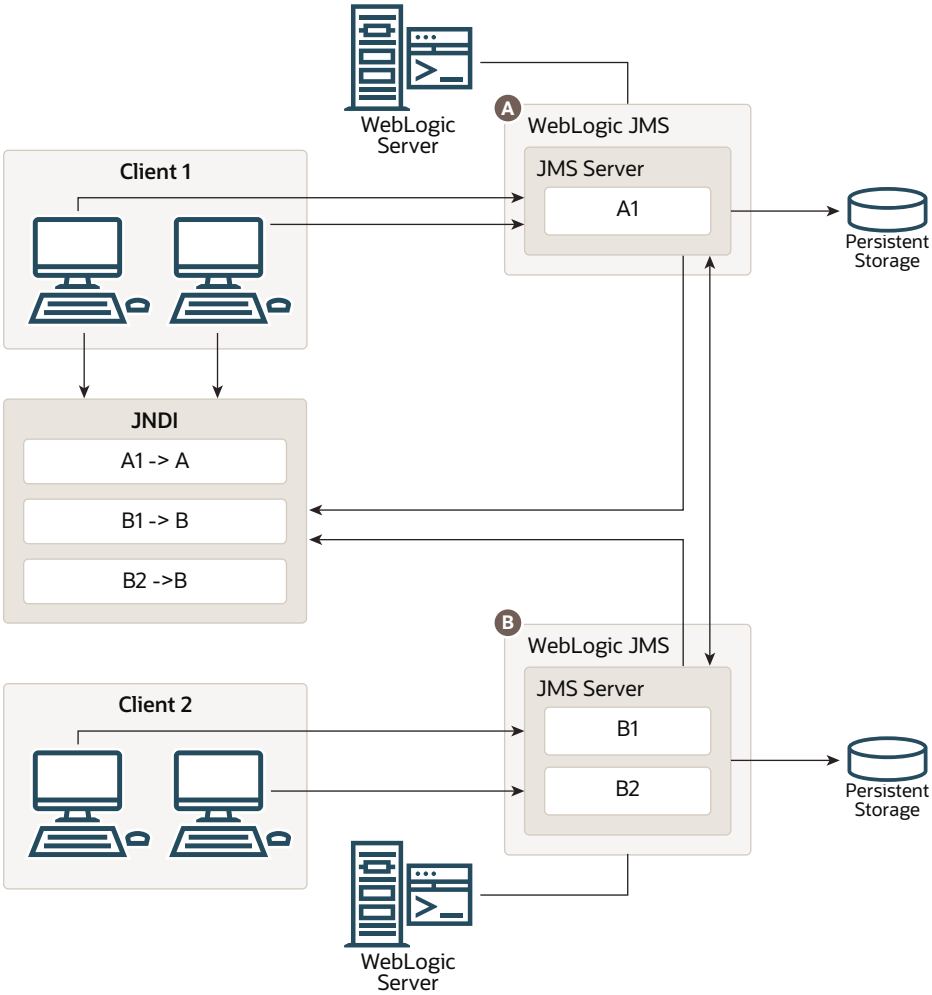
WebLogic Server is compliant with the following Java specifications.

- WebLogic Server is compliant with the Java Platform, Enterprise Edition (Java EE) specification, described at `https://javaee.github.io/javaee-spec/javadocs/`.
- WebLogic Server is fully compliant with the JMS 2.0 and JMS 1.1 specifications, at `http://www.oracle.com/technetwork/java/jms/index.html`, and can be used in production.

WebLogic JMS Architecture

Figure 2-2 illustrates the WebLogic JMS architecture.

Figure 1-2 WebLogic JMS Architecture



The major components of the WebLogic JMS Server architecture include:

- JMS server: a managed message container for a set of JMS queues and topics. Destination configuration is located in JMS XML modules that can target one or more JMS servers, and a single logical destination can be distributed across multiple JMS servers. A JMS server's primary responsibility for its targeted destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations. You can configure one or more JMS servers per domain, multiple JMS servers may run on the same WebLogic server, and a JMS server can manage one or more JMS modules.
- 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:

- The PTP messaging model enables the delivery of a message to exact one recipient.
- The publish/subscribe messaging model enables the delivery of a message to multiple recipients.

Each model is implemented with classes that extend common base classes. For example, the PTP class `javax.jms.Queue` (described at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Queue.html`) and the publish/subscribe class `javax.jms.Topic` (described at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Topic.html`) both extend the class `javax.jms.Destination` (described at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Destination.html`).

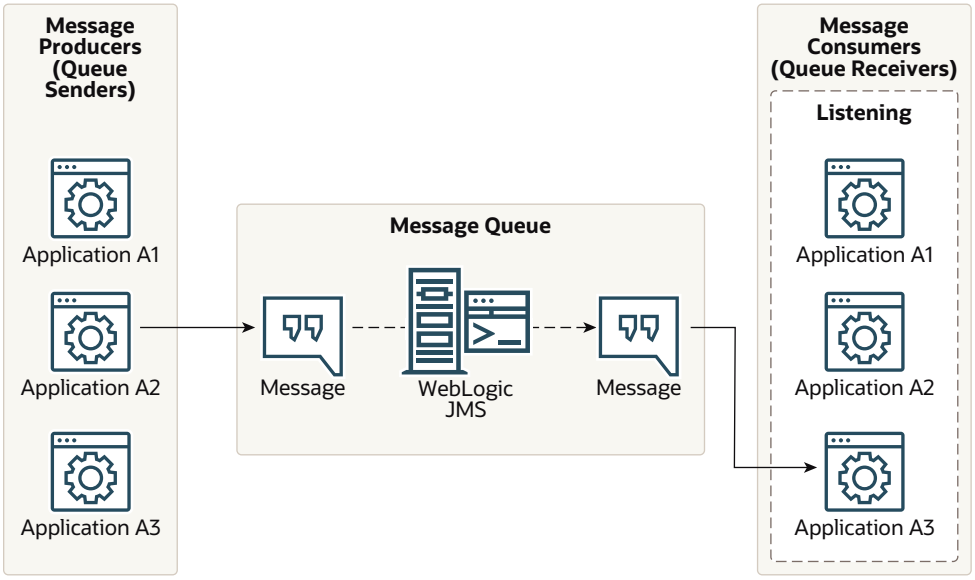
> **Note:**
>
> The terms *producer* and *consumer* are used as generic descriptions of applications that send and receive messages, respectively, in either messaging model. For each specific messaging model, however, unique terms specific to that model are used when referring to producers and consumers.

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.

Figure 2-3 illustrates PTP messaging.

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



Multiple queue senders and queue receivers can be associated with a single queue, but an individual message can be delivered to only *one* queue receiver.

If multiple queue receivers are listening for messages on a queue, then WebLogic JMS determines which one will receive the next message on a first come, first serve basis. If no queue receivers are listening on the queue, then messages remain in the queue until a queue receiver attaches to the queue.

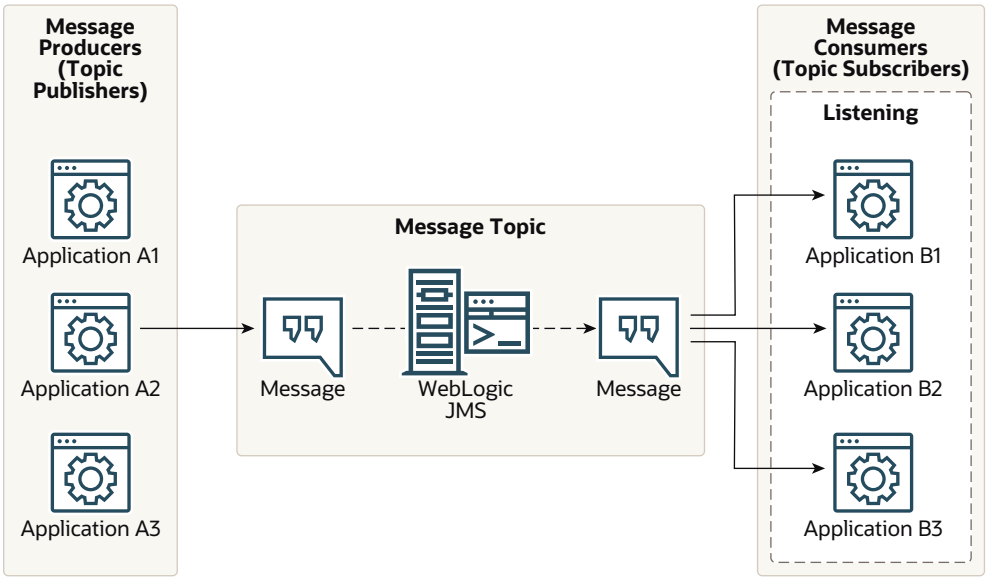
Publish/Subscribe Messaging

The publish/subscribe messaging model enables an application to send a message to multiple applications. Publish/subscribe messaging applications send and receive

messages by subscribing to a *topic*. A *topic publisher* (producer) sends messages to a specific topic. A *topic subscriber* (consumer) retrieves messages from a specific topic.

Figure 2-4 illustrates publish/subscribe messaging.

Figure 1-4 Publish/Subscribe Messaging



Unlike with the PTP messaging model, the publish/subscribe messaging model allows multiple topic subscribers to receive the same message. JMS retains the message until all topic subscribers have received it.

The publish/subscribe messaging model supports durable subscribers, enabling you to assign a name to a topic subscriber and associate it with a user or application. For more information about durable subscribers, see Setting Up Durable Subscriptions.

Message Persistence

The "Message Delivery Mode" section of the JMS Specification, described at `http://www.oracle.com/technetwork/java/jms/index.html`, messages can be specified as persistent or non persistent:

- A persistent message is guaranteed to be delivered *once*. The message cannot be lost due to a JMS provider failure, and it must not be delivered twice. It is not considered sent until it has been safely written to a file or database. WebLogic JMS writes persistent messages to a WebLogic persistent store (disk-base file or JDBC-accessible database) that is optionally targeted by each JMS server during configuration.
- Non persistent messages are not stored. They are guaranteed to be delivered *once-at-most-after*, unless there is a JMS provider failure, in which case messages may be lost, and must not be delivered twice. If a connection is closed or recovered, then all non persistent messages that have not yet been acknowledged will be redelivered. Once a non persistent message is acknowledged, it will not be redelivered.

For information about using the system-wide, WebLogic Persistent Store, see *Administering the WebLogic Persistent Store*.

Value-Added Public JMS API Extensions

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

In addition to fully supporting XA transactions, WebLogic JMS also features high availability through its clustering and service migration features, while also providing seamless interoperability with other versions of WebLogic Server and third-party messaging providers.

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

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 includes the classes and methods described in the Table 2-1.For more information about these APIs, see *Java API Reference for Oracle WebLogic Server*.

Table 1-1 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 • CreateJMS stream messages • Create JMS text messages • Create JMS XML messages
JMSMessageInfo	Provides browsing and message manipulation using JMX
JMSModuleHelper, JMSNamedEntityModifier	Monitors JMS runtime MBeans and manages JMS Module configuration entities in a JMS module
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	Sets a delivery time for messages, redelivery limits, and send timeouts
*Java API Reference for Oracle WebLogic Server*WLMessageProducer	Sets a message delivery times for producers and Unit-of-Order names

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

Interface/Class	Function
WLJMSContext	Provides additional fields and methods that are not supported by `javax.jms.JMSContext`. WLJMSContext provides the same extension features as WLConnection and WLSession
WLJMSProducer	Provides additional methods that are not supported by `javax.jms.JMSProducer`.
WLQueueSession, WLSession, WLTopicSession	Provides additional fields and methods that are not supported by `javax.jms.QueueSession`, `javax.jms.Session`, and `javax.jms.TopicSession`
XMLMessage	Creates XML messages
Schedule	Sets a scheduled delivery times for messages
JMSHelper	Monitors JMS runtime MBeans. Deprecated in this release of WebLogic Server. Replaced by 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 Java EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are a required part of Java 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.

Understanding the JMS API

The `javax.jms` API enables you to create the class objects necessary to connect to the JMS, and to send and receive messages.

To create a JMS application, use the `javax.jms` API at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/package-summary.html`. JMS class interfaces are created as subclasses to provide queue specific and topic specific versions of the common parent classes.

The Table 2-2 lists the JMS classes described in more detail in subsequent sections. For a complete description of all JMS classes, see `javax.jms`, at `https://javaee.github.io/`

javaee-spec/javadocs/javax/jms/package-summary.html, or in the weblogic.jms.extensions Javadoc.

Table 1-2 WebLogic JMS Classes

JMS Class	Description
ConnectionFactory	Encapsulates connection configuration information. A connection factory is used to create connections. You look up a connection factory using 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	Defines a serial order for the messages produced and consumed.
Destination	Identifies a queue or topic, encapsulating the address of a specific provider. Queue and topic destinations manage the messages delivered from the PTP and publish/subscribe messaging models, respectively.
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.
ConnectionConsumer[4]	Specifies a consumer that retrieves server sessions to process messages concurrently.

1 Supports an optional JMS interface for processing multiple messages concurrently.
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 JMS applications to create a `Connection` (see Connection). 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 supports the default connection factory as defined by the Java EE 7 specification. See Using the Default JMS Connection Factory Defined by Java EE 7 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 a 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 1-3 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.

For more information about the XA Connection Factory Enabled attribute, and to see the default values for the other connection factory attributes, see JMS Connection Factory: Configuration: Transactions in the *Oracle WebLogic Server Administration Console Online Help*.

Another distinction when using the default connection factories is that you have no control over targeting the WebLogic Server instances where the connection factory may be deployed. However, you can disable the default connection factories on a per-server basis.

For more information about enabling or disabling the default connection factories, see Servers: Configuration: Services in the *Oracle WebLogic Server Administration Console Online Help*.

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 `javax.jms.QueueConnectionFactory` and `javax.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 connection factories in the *Oracle WebLogic Server Administration 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 Class

The `ConnectionFactory` class does not define methods; however, its subclasses define methods for the respective messaging models. A connection factory supports concurrent use, enabling multiple threads to access the object simultaneously.

> **Note:**
>
> For this release, you can use the JMS version 1.1 specification connection factories or you can choose to use the subclasses.

Table 2-4 describes the `ConnectionFactory` subclasses.

Table 1-4 ConnectionFactory Subclasses

Subclass	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.

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

at https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionFactory.html.

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. In the WebLogic Server, JMS traffic is multiplexed with other WebLogic services on the client connection to the server. No additional TCP/IP connections are created for JMS. 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:**
>
> For this release, you can use the JMS Version 1.1 specification connection objects or you can choose to use the subclasses.

Table 2-5 describes the `Connection` subclasses.

Table 1-5 Connection Subclasses

Subclass	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` class within an application, see Developing a Basic JMS Application, or the `javax.jms.Connection` Javadoc at https://javaee.github.io/javaee-spec/javadocs/javax/jms/Connection.html.

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

The JMS 1.1 Specification, at `http://www.oracle.com/technetwork/java/jms/index.html`, allows for a generic session to have a `MessageConsumer` for any type of Destination object. However, WebLogic JMS does not support having both types of `MessageConsumer` (`QueueConsumer` and `TopicSubscriber`) for a single session. In addition, having multiple consumers for a single session is not a common practice. The following commonly used scenarios are supported:

- Using a single session with both a `QueueSender` and a `TopicSubscriber` or: `QueueConsumer` and `TopicPublisher`.
- Multiple `MessageProducer`s of any type.

> **Note:**
>
> A session and its message producers and consumers can only be accessed by one thread at a time. Their behavior is undefined if multiple threads access them simultaneously.

Session Subclasses

Table 2-6 describes the Session subclasses.

Table 1-6 Session Subclasses

Subclass	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 class within an application, see Developing a Basic JMS Application, or the `javax.jms.Session` at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Session.html`, and the weblogic.jms.extensions.WLSession Javadoc.

Non-Transacted Sessions

In a non-transacted session, the application creating the session selects one of the five acknowledge modes defined in Table 2-7.

Table 1-7 Acknowledge Modes Used for Non-Transacted Sessions

Acknowledge Mode	Description
`AUTO_ACKNOWLEDGE`	The `Session` object acknowledges receipt of a message after receiving application method has returned from processing it.
`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. **Note:** In the WebLogic Server Administration Console, if the Acknowledge Policy attribute on the connection factory is set to `Previous`, but you want to acknowledge *all* received messages for a given session, then use the last message to invoke the acknowledge method. For more information on the Acknowledge Policy attribute, see JMS Connection Factory: Configuration: General in the *Oracle WebLogic Server Administration Console Online Help*.
`DUPS_OK_ACKNOWLEDGE`	The `Session` object acknowledges receipt of a message after the receiving application method has returned from processing it; duplicate acknowledges are permitted. This mode is most efficient in terms of resource usage. **Note:** You should avoid using this mode if your application cannot handle duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.
`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.

Table 1-7 (Cont.) Acknowledge Modes Used for Non-Transacted Sessions

Acknowledge Mode	Description
`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 transacted session, only one transaction is active at any 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.

JMS can participate in distributed transactions with other Java services, such as EJB, that use the Java Transaction API (JTA). Transacted sessions do not support this capability because the transaction is restricted to accessing the messages associated with that session. For more information about using JMS with JTA, 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.

A destination supports concurrent use, enabling multiple threads to access the object simultaneously. JMS `Queue` and `Topic` objects extend `javax.jms.Destination` method described at `http://docs.oracle.com/javaee/7/api/javax/jms/Destination.html`.

> **Note:**
>
> For this release, you can use the JMS version 1.1 specification destination objects or you can choose to use the subclasses.

Table 2-8 describes the `Destination` subclasses.

Table 1-8 Destination Subclasses

Subclass	Messaging Model	Manages Messages for
`Queue`	PTP	JMS point-to-point provider.
`TemporaryQueue`	PTP	JMS point-to-point provider, and exists for the duration of the JMS connection in which the messages are created. A temporary queue can be consumed only by the queue connection that created it
`Topic`	Pub/sub	JMS publish/subscribe provider
`TemporaryTopic`	Pub/sub	JMS publish/subscribe provider, and exists for the duration of the JMS connection in which the messages are created. A temporary topic can be consumed only by the topic connection that created it

> **Note:**
>
> An application has the option of browsing queues by creating a `QueueBrowser` object in its queue session. This object produces a snapshot of the messages in the queue at the time the queue browser is created. The application can view the messages in the queue, but the messages are not considered read and are not removed from the queue. For more information about browsing queues, see Setting and Browsing Message Header and Property Fields .

To learn how to use the `Destination` class within an application, see Developing a Basic JMS Application, or the `javax.jms.Destination` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Destination.html`.

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. Applications that use a distributed destination are more highly available than applications that use standalone destinations because WebLogic JMS provides load balancing and failover for the members of a distributed destination in a cluster.

- For more information about using a distributed destination with your applications, see Using Distributed Destinations.
- For instructions about configuring a distributed queue destination, see Configure uniform distributed queues in the *Oracle WebLogic Server Administration Console Online Help*.
- For instructions about configuring a distributed topic destination, see Configure uniform distributed topics in the *Oracle WebLogic Server Administration Console Online Help*.

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 message sender and receiver objects are created as subclasses of the `MessageProducer` and `MessageConsumer` classes.

> **Note:**
>
> For this release, you can use the JMS version 1.1 specification message producer and consumer objects or you can use the subclasses.

Table 2-9 describes the `MessageProducer` and `MessageConsumer` subclasses.

Table 1-9 MessageProducer and MessageConsumer Subclasses

Subclass	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

The PTP model, as shown in the figure Figure 1-3, allows multiple sessions to receive messages from the same queue. However, a message can only be delivered to one

queue receiver. When there are multiple queue receivers, WebLogic JMS defines the next queue receiver that will receive a message on a first-come, first-serve basis.

The Publish/Subscribe model, as shown in the figure Figure 1-4, allows messages to be delivered to multiple topic subscribers. Topic subscribers can be durable or non-durable, as described in Setting Up Durable Subscriptions.

An application can use the same JMS connection to both publish and subscribe to a topic. Because topic messages can be delivered to all subscribers, an application can receive messages it has published itself. To prevent clients from receiving messages that they publish, a JMS application can set a `noLocal` attribute on the topic subscriber, as described in Step 5: Create Message Producers and Message Consumers.

To learn how to use the `MessageProducer` and `MessageConsumer` classes within an application, see Setting Up a JMS Application, or the `javax.jms.MessageProducer` (at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageProducer.html`), and `javax.jms.MessageConsumer` (at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageConsumer.html`) 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 `javax.jms.Message` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html`.

Table 2-10 describes the fields in the message headers and shows how values are defined for each field.

Table 1-10 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 messaging. This field is set on the producer or as parameter sent by the application before send(). When a persistent message is sent, it is stored in the WebLogic Persistent Store. The send() operation is not considered successful until delivery of the message can be guaranteed. A persistent message is guaranteed to be delivered at least once. WebLogic JMS does not store non-persistent messages in the persistent store. This mode of operation provides the lowest overhead. They are guaranteed to be delivered at least once unless there is a system failure, in which case messages may be lost. If a connection is closed or recovered, all non persistent messages that have not yet been acknowledged will be redelivered. After a non persistent message is acknowledged, it will not be redelivered. This value is overwritten by a call to theproducer.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.	send() method
JMSDeliveryTime	Defines the earliest absolute time at which a message can be delivered to a consumer. This field is set by the application before send() and depends on timeToDeliver, which is set on the producer. 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

Table 1-10 (Cont.) Message Header Fields

Field	Description	Defined by
`JMSExpiration`	Specifies the expiration, or time-to-live value, for a message. This field is set by the application before `send()`. Depends on `timeToLive`, which is set on the producer or as a parameter sent by the application to `send()`. WebLogic JMS calculates the `JMSExpiration` value as the sum of the application's time-to-live and the current GMT. If the application specifies time-to-live as 0, then the `JMSExpiration` value is set to 0, which means the message never expires. WebLogic JMS removes expired messages from the system to prevent their delivery.	`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. You can sort destinations by priority by configuring a destination key, as described in Configure destination keys in the *Oracle WebLogic Server Administration Console Online Help*.	`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

Table 1-10 (Cont.) Message Header Fields

Field	Description	Defined by
`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
`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 `javax.jms.Message` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html`. 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 `http://www.oracle.com/technetwork/java/jms/index.html`:

Table 1-11 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 http://www.oracle.com/technetwork/java/jms/index.html

Message Body

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

Table 2-12 describes the types of messages defined by JMS. All message types extend javax.jms.Message, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html, which consists of message headers and properties, but no message body.

Table 1-12 JMS Message Types

Type	Description
javax.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 https://javaee.github.io/javaee-spec/javadocs/javax/jms/BytesMessage.html.
javax.jms.MapMessage	Set of name/value pairs in which the names are strings and the values are Java primitive types. Pairs can be read sequentially or randomly, by specifying a name.
javax.jms.ObjectMessage	Single serializable Java object. See https://javaee.github.io/javaee-spec/javadocs/javax/jms/ObjectMessage.html.
javax.jms.StreamMessage	Similar to a BytesMessage, except that only Java primitive types are written to or read from the stream. See https://javaee.github.io/javaee-spec/javadocs/javax/jms/StreamMessage.html.
javax.jms.TextMessage	Single String. The TextMessage can also contain XML content. See https://javaee.github.io/javaee-spec/javadocs/javax/jms/TextMessage.html.
weblogic.jms.extensions.XMLMessage	XML content. Use of the XMLMessage type facilitates message filtering, which is more complex when performed on XML content shipped in a TextMessage. See XMLMessage.

For more information, see the javax.jms.Message Javadoc at https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html. 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 Java 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 Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

A server session pool is a WebLogic-specific JMS feature that enables you to process messages concurrently. A server session pool factory is used to create a server-side `ServerSessionPool`.

WebLogic JMS defines one `ServerSessionPoolFactory` object, by default: `weblogic.jms.extensions.ServerSessionPoolFactory:<name>`, the <*name*> specifies the name of the JMS server to which the session pool is created. The WebLogic Server adds the default server session pool factory to the JNDI space during startup and the application subsequently retrieves the server session pool factory using WebLogic JNDI.

To learn how to use the server session pool factory within an application, see Defining Server Session Pools, or the weblogic.jms.extnesions.ServerSessionPoolFactory Javadoc.

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.

To learn how to use the server session pool within an application, see Defining Server Session Poolsor the `javax.jms.ServerSessionPool` application Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ServerSessionPool.html`.

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.

To learn how to use the server session within an application, see Defining Server Session Pools or the `javax.jms.ServerSession` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ServerSession.html`.

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.

To learn how to use the connection consumers within an application, see Defining Server Session Pools, or the `javax.jms.ConnectionConsumer` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionConsumer.html`.

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

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`, 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 Java strings is more expensive than serializing other Java primitive types. Strings are also memory intensive; they consume two bytes of memory per Character, and cannot compactly represent binary data (integers, for example). 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

WebLogic JMS servers do not incur the cost of serializing non persistent messages. Serialization of non persistent message types is incurred by the remote client. Persistent messages are serialized by the server.

Selection

Using a selector is expensive. This consideration is important when you are deciding where in the message to store application data that is accessed through JMS selectors.

Message Compression

Compressing large messages in a JMS application can improve performance.

Message compression reduces the amount of time required to transfer messages across the network, reduces the amount of memory used by the JMS server, and, if the messages are persistent, reduces the size of persistent writes. Text and XML messages can often be compressed significantly. Of course, compression is achieved at the expense of an increase in the CPU usage of the client.

Keep in mind that the benefits of compression become questionable for smaller messages. If a message is less than a few KB in size, then compression can actually increase its size. The JDK provides built-in compression libraries. For details, see 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

Instead of user-defined message properties, consider using standard JMS message header fields or the message body for message data. Message properties incur an extra cost in serialization, and are more expensive to access than standard JMS message header fields.

Avoid embedding large amounts of data in the properties field or the header fields; only message bodies are paged out when paging is enabled. Consequently, if user defined message properties are defined in an application, avoid the use of large string properties.

See Message Header Fields and Message Property Fields .

Message Ordering

You should use the Message Unit-of-Order feature rather than Ordered Redelivery to guarantee ordered message processing.

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 `MessagingMaximum` 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

When you start to design your application, it is not always immediately obvious whether it would be better to use a Topic or Queue.

You should use a Topic only if one of the following conditions applies:

- The same message must be replicated to multiple consumers.
- A message should be dropped if there are no active consumers that will select it.
- There are many subscribers, each with a unique selector.

Note that a topic with a single durable subscriber is semantically similar to a queue. The differences are as follows:

- If you change a topic selector for a durable subscriber, then all previous messages in the subscription are deleted, while if you change a queue selector for consumer, then no messages in the queue are deleted.
- A queue may have multiple consumers, and will distribute its messages in a round-robin fashion, whereas a topic subscriber is limited to one consumer.

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

When designing an application, make sure you specify that messages will be sent in non persistent mode unless a persistent QOS is required.

Oracle recommends non persistent mode because unless synchronous writes are disabled, a persistent QOS can cause a significant degradation in performance.

> **Note:**
>
> Avoid persisting sending persistent messages unintentionally. Occasionally an application sends persistent messages even though the designer intended the messages to be sent in non persistent mode.

If your messages are truly non persistent, none should end up in a regular JMS store. To make sure that none of your messages are persistent, check whether the JMS store size grows when unconsumed messages are accumulating on the JMS server. Here is how message persistence is determined, in order of precedence:

- Producer's connection's connection factory configuration:
  - PERSISTENT (default)
  - NON_PERSISTENT
- JMS Producer API override on QueueSender and TopicPublisher:

  - setDeliveryMode(DeliveryMode.PERSISTENT)
  - setDeliveryMode(DeliveryMode.NON_PERSISTENT)
  - setDeliveryMode(DeliveryMode.DEFAULT_DELIVERY_MODE) (default)
- JMS Producer API per message override on QueueSender and TopicPublisher:
  - For queues, optional deliveryMode parameter on send()
  - For topics, optional deliveryMode parameter on publish()
- Override on destination configuration:
  - Persistent
  - Non Persistent
  - No Delivery (default, implies no override)
- Override on JMS server configuration:
  - If store is configured then that implies using the default persistent store that is available on each targeted WebLogic Server instance
  - If a Store is configured then that implies no override.
- Non durable subscribers only:
  - If there are no subscribers, or there are only non durable subscribers for a topic, the messages will be downgraded to non persistent. (Because non durable subscribers exist only for the life of the JMS server, there is no reason for the message to persist.)
- Temporary destinations:
  - Because temporary destinations exist only for the lifetime of their host JMS server, there is no reason for messages to persist. WebLogic JMS automatically forces all messages in a temporary destination to non-persistent.

Durable subscribers require a persistent store to be configured on their JMS server, even if they receive only non persistent messages. A durable subscription persists to ensure that it continues through a server restart, as required by the JMS specification.

Deferring Acknowledges and Commits

Because sending is generally faster than receiving , consider reducing the overhead associated with receiving by deferring acknowledgment of messages until several messages have been received and can be acknowledged collectively.

If you are using transactions, then substitute the word `commit` for `acknowledge`.

Deferment of acknowledgements is not likely to improve performance for non durable subscriptions, because of the internal optimizations already in place.

It may not be possible to implement deferred acknowledgements for asynchronous listeners. If an asynchronous listener acknowledges only every 10 messages, but for some reason receives only 5, then the last few messages may not be acknowledged. One possible solution is to have the asynchronous consumer post synchronous, non blocking receives from within its `onMessage()` callback to receive subsequent messages. Another possible solution is to have the listener start a timer that, when triggered, sends a message to the listener's destination in order to wake it up and complete the outstanding work that has not yet been acknowledged—assuming that the wake-up message can be directed at the correct listener.

Using AUTO_ACK for Non Durable Subscribers

Non durable, non transactional topic subscribers are optimized to store local copies of the message on the client side, thus reducing network overhead when acknowledgements are being issued.

This optimization yields a 10-20 percent performance improvement, where the improvement is more evident under higher subscriber loads.

One side effect of this optimization, particularly for high numbers of concurrent topic subscribers, is the overhead of client-side garbage collection, which can degrade performance for message subscriptions. To prevent such degradation, Oracle recommends allocating a larger heap size on the subscriber client. For example, in a test of 100 concurrent subscribers running in 10 JVMs, it was found that giving clients an initial and maximum heap size of 64MB for each JVM was sufficient.

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
- Using NO_ACKNOWLEDGE

Using MULTICAST_NO_ACKNOWLEDGE

Non durable topic subscribers can subscribe to messages using the `MULTICAST_NO_ACKNOWLEDGE`. If a topic has such subscribers, then the JMS server will broadcast messages to them using multicast mode. Multicast improves performance considerably and provides linear scalability, as the network only needs to handle one message, regardless of the number of subscribers, rather than one message per subscriber. Multicast messages may be lost if the network is congested or if the client falls behind in processing them. Calls to `recover()` or `acknowledge()` have no effect on multicast messages.

> **Note:**
>
> On the client side, each multicasting session requires a dedicated thread to retrieve messages off the multicast socket. Therefore, you should increase the JMS client-side thread pool size to adjust for this.

This QOS extension has the same level of guarantee as some JMS implementations default QOS from vendors other than Oracle WebLogic Server for non durable topic subscriptions. The JMS 1.1 specification specifically allows non durable topic messages to be dropped (deleted) if the subscriber is not ready for them. WebLogic JMS has a higher QOS for non durable topic subscriptions by default than the JMS 1.1 specification requires.

Using NO_ACKNOWLEDGE

A no-acknowledge delivery mode implies that the server gives messages to consumers, but does not expect an acknowledgement to be called. Instead, the server pre-acknowledges the message. In this acknowledge mode, calls to recover will not work, because the message was acknowledged. This mode saves the overhead of an additional network call to the acknowledge, at the expense of possibly losing a message when a server failure, a network failure, or a client failure occurs.

> **Note:**
>
> If an asynchronous client calls the `close()` in this scenario, then all messages in the asynchronous pipeline are lost.

Asynchronous consumers that use a `NO_ACKNOWLEDGE` QOS may want to reduce their message pipeline size in order to lower the number of lost messages in the event of a failure.

Avoid Multi threading

The JMS specification states that multi threading a session, producer, consumer, or message method results in undefined behavior except when calling `close()`.

See the specification at `http://www.oracle.com/technetwork/java/jms/index.html`. If your application is thread limited, then try increasing the number of producers and sessions.

Using the JMSXUserID Property

For WebLogic Server 9.0 and later, you can configure a JMS connection factory and destination to automatically propagate the message sender's authenticated username. The username is placed in a `javax.jms.Message` property named `JMSXUserID`.

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

- While the JMS specification makes some mention of the `JMSXUserID` property, the behavior is lightly defined and will likely be different for different JMS vendors.
- The `JMSXUserID` property is based on the credential of the thread an application uses to create the JMS producer. It does not derive from the credential that is on a thread during the JMS send call itself.
- JMS will ignore or override any attempt by an application to directly set `JMSXUserID` (for example, `javax.jms.Message.setXXXProperty()` will not work).
- JMS messages are not signed or encrypted (similar to any RMI/EJB call). Therefore, fully secure transfers of the `JMSXUserID` require sending the message through secure protocols (for example, `t3s` or `https`).
- WebLogic Store-and-Forward agents do not propagate the `JMSXUserID` (they null it out).
- WebLogic Messaging bridges will propagate `JMSXUserID` property of the source destination's message if the messaging bridges are both are forwarding to a 9.0 or later JMS server and are configured to **Preserve Message Properties**. Otherwise, the

forwarded message will either contain no username or the username used by the bridge sender. The latter behavior is determined by the configuration of the bridge sender's connection factory and destination.

- The WebLogic JMS `WLMessageProducer.forward()` extension can forward a received message's `JMSXUserID`.

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

For instructions about setting the `JMSXUserID` property on a connection factory or a destination, see the following topics in the WebLogic Server Administration Console online help:

- Configure connection factory security parameters
- Configure advanced queue parameters
- Configure advanced topic parameters
- Uniform distributed queues - configure advanced parameters
- Uniform distributed topics - configure advanced parameters

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*.

3

Enhanced Support for Using WebLogic JMS with EJBs and Servlets

Learn about WebLogic Server enhancements, such as JMS wrappers, that extend the Java EE standard to make it easier to access EJB and servlet containers with WebLogic JMS or third-party JMS providers. Implementing JMS wrapper support is the best practice method of how to send a WebLogic JMS message from inside an EJB or servlet.

Enabling WebLogic JMS Wrappers

WebLogic Server uses JMS wrappers that make it easier to use WebLogic JMS inside a Java 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

WebLogic Server 12.2.1 release supports the JMS 2.0 simplified API, which enables you to inject a `JMSContext` object into the application using the `@Inject` annotation as follows:

```
@Inject
@JMSConnectionFactory("myJMSCF")
```

```
@JMSPasswordCredential(   userName="admin",    password="admin_password")private
JMSContext context;
```

The `@Inject` annotation determines when the container should create the `JMSContext` object.

> **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 Java EE Platform in *Developing Applications for Oracle WebLogic Server*.
> - If the injected `JMSContext` is null and if your application fails, then review the server log. If the connection factory could not be found, you can see that error in the server log. If there is no error in the server log then the application failure is probably due to a missing `beans.xml` file.

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, the resource reference name must be defined using a `<resource-ref>` element in the deployment descriptor that maps it to an appropriate product-specific global JNDI name. See Declaring a Wrapped JMS Factory using Deployment Descriptors.

If no lookup name is provided for the `@JMSConnectionFactory` annotation, then the Java EE platform default JMS connection factory (`java:comp/DefaultJMSConnectionFactory`) will be used.

Determining the Authentication Type for JMSContext Injection

The JMSContext injection cannot use the resource reference to determine whether the connection factory should use container authentication or application authentication. Instead, you can use the `@JMSPasswordCredential` annotation to specify the type of authentication required.

If you specify the `@JMSPasswordCredential` annotation then the connection factory will use password authentication, and the specified user and password. If the `@JMSPasswordCredential` annotation is not defined then the connection factory will use container authentication.

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

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

- Declaring a Wrapped JMS Factory using Deployment Descriptors
- Declaring JMS Destinations using Deployment Descriptors

For more information about packaging EJBs, see Implementing Enterprise JavaBeansin *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*. 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>javax.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.

In addition to this element, there must be a matching `resource-description` element in the `ejb-jar.xml` (for EJBs) or `weblogic.xml` (for servlets) file that tells the Java EE container which JMS connection factory to put in that location. Here is an example:

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

The connection factory specified here must already exist in the global JNDI tree. (This example uses one of the default JMS connection factories that is automatically created when the built-in WebLogic JMS server is used). To use another WebLogic JMS connection factory from the same cluster, include that connection factory's JNDI name inside the `jndi-name` element. To use a connection factory from another vendor, or from another WebLogic Server cluster, create a Foreign JMS Server.

If the JNDI name specified in the `resource-description` element is incorrect, then the application is still deployed. However, you will receive an error when you try to use the connection factory.

Declaring JMS Destinations using Deployment Descriptors

You can define a JMS destination resource in a web module, EJB module, application client module, or in an application deployment descriptor using the `jms-destination` or `resource-env-ref` descriptor elements.

> **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>javax.jms.Queue</interface-name>
  <destination-name>myQueue1</destination-name>
  <property>
    <name>Property1</name>
    <value>10</value>
  </property>
  <property>
    <name>Property2</name>
    <value>20</value>
  </property>
</jms-destination>
```

For more information about the `jms-destination` element and its attributes, see the schema at http://xmlns.jcp.org/xml/ns/javaee/javaee_7.xsd.

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.

For `resource-env-ref` description, the queue or topic destination specified in the descriptor must already exist in the global JNDI tree. Again, if the destination does not exist, then the application is deployed, but an exception is thrown when you try to use 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>javax.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 Java 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 Java 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 Java 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>
```

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 Java EE Application components.

Referencing JMS Resources in a Java EE Application

The `name` element of a JMS connection factory resource specified in the JMS module must match the `res-ref-name` element defined in the referring EJB or WebApp application descriptor file. The `res-ref-name` element maps the resource name (used by `java:comp/env`) to a module referenced by an EJB.

For queue or topic destination resources specified in the JMS module, the `name` element must match the `resource-env-ref` field defined in the referring module descriptor file.

That name is how the link is made between the resource referenced in the EJB or web application module and the resource defined in the JMS module. For example:

```
<resource-ref>
  <res-ref-name>jms/OrderQueueFactory</res-ref-name>
  <res-type>javax.jms.ConnectionFactory</res-type>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>jms/OrderQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.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 Enterprise JavaBeans for Oracle WebLogic Server*.

Injecting Resource Dependency into a Class

If you apply the @Resource to a class, then the resource is made available in the comp/env context. The following is an example of how to inject a WebLogic JMS destination and connection factory resource in a Java EE application, including EJBs, MDBs, and servlets.

Example 4-1 is a Wrapped JMS Pooling Annotation example:

Example 3-1 Wrapped JMS Pooling Annotation Example

```
.
.
.
// The "name=" or "type=" are not always required,
// "mappedName=" is usually sufficient.
@Resource(name="ReplyQueue",
 type=javax.jms.Queue.class,
 mappedName="jms/ReplyQueue") Destination rq;
.
.
.
@Resource(name="ReplyConnectionFactory",
 type=javax.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 4-2:

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=javax.jms.ConnectionFactory.class),

  @Resource(name="targetDestRef",
            mappedName="TargetDestJNDIName",
            type=javax.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() {

      // 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 =
        (javax.jms.ConnectionFactory)sctx.lookup("targetCFRef");

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

Avoid Transactional XA Interfaces

With resource wrapping, do not use the `javax.jms` XA transactional 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.class` 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 Java 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 Java EE behavior, as described in the following sections.

- Automatically Enlisting Transactions
- Container-Managed Security
- Connection Testing
- Java EE Compliance
- Pooled JMS Connection Objects
- Monitoring Pooled Connections

Automatically Enlisting Transactions

Automatically Enlisting Transaction works for either WebLogic JMS implementations or for third-party JMS providers that support two-phase commit transactions (XA protocol). If a wrapped JMS connection sends or receives a message inside a transaction context, then the JMS session being used to send or receive the message is automatically enlisted in the transaction through the XA capabilities of the JMS provider. This is the case whether the transaction was started implicitly because the JMS code was invoked inside an EJB with container-managed transactions enabled, or whether the transaction was started manually using the `UserTransaction` interface in a servlet or an EJB that supports bean-managed transactions.

However, if an EJB or servlet attempts to send or receive a message inside a transaction context and the JMS provider does not support XA, the `send()` or `receive()` call throws the following exception:

```
[J2EE:160055] Unable to use a wrapped JMS session in the transaction because two-phase
commit is not available.
```

Therefore, if you are using a JMS provider that doesn't support XA to send or receive a message inside a transaction, then either declare the EJB with a transaction mode of `NotSupported` or suspend the transaction using one of the JTA APIs.

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 Java 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:**
>
> When you inject a `JMSContext` object into the application and if the JNDI name of the connection factory is specified by `@JMSConnectionFactory`, then container authentication is used. If you specify the username and password in the `@JMSPasswordCredential` annotation to specify the user/password, application authentication is used. See Declaring a JMSContext Object Using @Inject Annotation.

Connection Testing

The JMS wrapper classes monitor each connection that is established to the JMS provider. They do this in two ways:

- Registering a JMS `ExceptionListener` object on the connection.
- Testing the connection every 2 minutes by sending a message to a temporary queue or topic and then receiving it again.

Java EE Compliance

The Java EE specification states that you should not be allowed to make certain JMS API calls inside a Java EE application. The JMS wrappers enforce these restrictions by throwing the following exceptions when they are violated:

- On the connection object, the methods `createConnectionConsumer()`, `createDurableConnectionConsumer()`, `setClientID()`, `setExceptionListener()`, and `stop()` should not be called.
- On the session object, the methods `getMessageListener()` and `setMessageListener()` should not be called.
- On the consumer object (a `QueueReceiver` or `TopicSubscriber` object), the methods `getMessageListener()` and `setMessageListener()` should not be called.

Furthermore, the `createSession()` method, and the associated `createQueueSession()` and `createTopicSession()` methods, are handled differently. The `createSession()` method takes two parameters: an "acknowledgement" mode

and a "transacted" flag. When used inside an EJB, these two parameters are ignored. If a transaction is present, then the JMS session is enlisted in the transaction as described in Automatically Enlisting Transactions; otherwise, it is not. By default, the acknowledgement mode is set to "auto acknowledge". This behavior is expected by the Java EE specification.

> **Note:**
>
> This may make it more difficult to receive messages from inside an EJB, but the recommended way to receive messages from inside an EJB is to use a MDB, as described in *Developing Message-Driven Beans for Oracle WebLogic Server*.

Inside a servlet, however, the parameters to `createQueueSession()` and `createTopicSession()` are handled normally, and users can make use of all the various message acknowledgement modes.

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 Java 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.

Monitoring Pooled Connections

You can use the WebLogic Server Administration Console to monitor pooled connections. For more information, see JMS Servers: Monitoring: Active Pooled Connections in the *Oracle WebLogic Server Administration Console Online Help*.

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 inSending a JMS Message in a Java 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
- Speeding Up Object Creation Through Caching
- Enlisting the Proper Transaction Mode

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 Java EE specifications), because the number of a times that lookups 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 Java EE specification.)

However, if these objects are cached inside the `ejbCreate()` or `init()` method, then the EJB or servlet must have some way to recreate them if was a failure. This is necessary because some JMS providers, like WebLogic JMS, may invalidate a Destination object after a server failure. So, if the EJB runs on *Server A*, and JMS runs on *Server B*, then the EJB on *Server A* must perform the JNDI lookup of the objects from *Server B* again after that server has recovered. The example, PoolTestBean.java includes a sample EJB that performs this caching and re-lookup process correctly.

Speeding Up Object Creation Through Caching

After Connection Factory object and Destination object pooling is established, it may be tempting to cache other objects, such as the Connection, Session, and Producer objects, inside the `ejbCreate()` method. This will work, but it is not always the most efficient solution. Essentially, by doing this you are removing a Session object from the cache and permanently assigning it to a particular EJB, whereas by using the JMS wrappers as designed, that Session object can be shared by other EJBs and servlets as well. Furthermore, the wrappers attempt to reestablish a JMS connection and create new session objects if there is a communication failure with the JMS provider, but this will not work if you cache the Session object on your own.

Enlisting the Proper Transaction Mode

When a JMS `send()` or `receive()` operation is performed inside a transaction, the EJB or servlet automatically enlists the JMS provider in the transaction. A transaction can be started automatically inside an EJB or servlet that has container-managed transactions, or it can be started explicitly using the `UserTransaction` interface. In either case, the container automatically enlists the JMS provider. However, if the underlying JMS connection factory used by the EJB or servlet does not support XA, then the container throws an exception.

Performing the transaction enlistment has overhead. Furthermore, if an XA connection factory is used, but the `send()` or `receive()` method is invoked outside a transaction, then the container must still create a JTA transaction to wrap the `send()` or `receive()` method in order to ensure that the operation properly takes place no matter which JMS provider is used. Although this is only a one-phase commit, it can still slow down the server.

Therefore, when writing an EJB or servlet that uses a JMS resource in a non-transactional manner, it is best to use a JMS connection factory that is not configured to support XA.

Simplified Access to Foreign JMS Providers

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

See Accessing Foreign JMS Providers in the *Administering JMS Resources for Oracle WebLogic Server*. This feature makes it possible to easily map foreign JMS providers — including remote instances of WebLogic Server in another cluster or domain — so that they appear in the local JNDI tree as a local JMS object.

Another set of foreign JMS provider features makes it possible to create a "symbolic link" between a JMS connection factory or destination object in an third-party JNDI provider to an object inside the local WebLogic Server. This feature can also be used to reference remote instances of WebLogic Server in another cluster or domain in the local WebLogic JNDI tree.

There are three System Module MBeans for this task:

- Foreign server : Contains information about the remote JNDI provider, including its initial context factory, URL, and additional parameters. It is the parent of the Foreign Connection Factory and Foreign Destination MBeans. It can be targeted to an independent WebLogic Server or to a cluster. For more information see, ForeignServerBean in the *MBean Reference for Oracle WebLogic Server*.
- Foreign connection factory : Represents a foreign connection factory. It contains the name of the connection factory in the remote JNDI provider, the name to map it to in the server's JNDI tree, and an optional user name and password. The user name and password are only used when a Foreign Connection Factory is used inside a `resource-reference` in an EJB or a servlet, with the "Container" mode of *authentication*. It creates non-replicated JNDI objects on each WebLogic Server instance to which the parent Foreign Connection Factory MBean is targeted. (To create the JNDI object on every node in a cluster, target the parent MBean to the cluster.). For more information see, ForeignConnectionFactoryBean in the *MBean Reference for Oracle WebLogic Server*.
- Foreign destination : Represents a foreign destination. It contains the name to look up on the foreign JNDI provider, and the name to map it to on the local server.

Examples of JMS Wrapper Functions

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

Examples of JMS Wrapper Functions

The following files make up a simple stateless EJB session bean that uses the WebLogic JMS wrapper functions to send a transactional message (`sendXATransactional`) when an EJB is called. Although this example uses a session bean, the same XML descriptors and bean class (with very few changes) can be used for a message-driven bean.

- ejb-jar.xml
- weblogic-ejb-jar.xml
- PoolTest.java
- PoolTestHome.java
- PoolTestBean.java

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`).

```
<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">
<?xml version="1.0"?>
...
<ejb-jar>
 <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>javax.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>javax.jms.Queue</resource-env-ref-type>
  </resource-env-ref>
 </session>
 </enterprise-beans>

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

weblogic-ejb-jar.xml

This section declares matching `resource-description` queue connection factory and queue destination elements that tell the Java EE container which JMS connection factory and destination to put in that location.

```
<!DOC<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-ejb-jar>
 <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>
   <res-env-ref-name>jms/TESTQUEUE</res-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 javax.ejb.*;
public interface PoolTest extends EJBObject
{
 public String sendXATransactional(String text)
  throws RemoteException;
}
```

PoolTestHome.java

This section defines the "home" interface for the `PoolTest` bean. It is required by the EJB specification.

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

import java.rmi.*;
import javax.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 javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import javax.transaction.*;

public class PoolTestBean
 extends PoolTestBeanBase
 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();
   }
 }
```

```
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, 0);
   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.
  // this is necessary because the destination object
  // can become invalid if the destination server has
  // been shut down.
  qcf = null;
  destination = null;
  throw new RemoteException("Failure in EJB: " + e);
 }
 return id;
}
}
```

Sending a JMS Message in a Java 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 4-3 sends a message if you map to the `java:comp/env` JNDI tree:

Example 3-3 Sending a Message Using comp/env

```
.
.
.
```

```
InitialContext ic = new InitialContext();
QueueConnectionFactory qcf =
 (QueueConnectionFactory)ic.lookup("java:comp/env/jms/QCF");
Queue destQueue =
 (Queue)ic.lookup("java:comp/env/jms/TESTQUEUE");
ic.close();
QueueConnection connection = qcf.createQueueConnection();
try {
 QueueSession session = connection.createQueueSession(0, false);
 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 Java EE specification and should run on any EJB or servlet product that properly supports Java 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 Java 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 using a *wrapped* JMS connection factory, which is obtained by using the `resource-ref` feature and looked up by using the `java:comp/env/jms` JNDI tree context, the EJB must not use the `javax.jms` XA transactional XA interfaces.

Dependency Injection

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

Example 3-4 Sending a Message using Dependency Injection

```
package test;
// Example injected annotation.
import javax.annotation.Resource;
```

```
import javax.ejb.*;
import javax.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(true, 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 4-5 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 javax.annotation.Resources;
import javax.annotation.Resource;
import javax.naming.*;
import javax.ejb.*;
import javax.jms.*;

import javax.ejb.ActivationConfigProperty;

@MessageDriven(
     name = "MyMDB",
     mappedName = "JNDINameOfMDBSourceDest",
     activationConfig = {
     // the JMS interface type for the MDB destination, either javax.jms.Topic or javax
.jms.Queue
     @ActivationConfigProperty(
```

```
          propertyName = "destinationType",
          propertyValue = "javax.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=javax.jms.ConnectionFactory.class),

  @Resource(name="targetDestRef",
            mappedName="TargetDestJNDIName",
            type=javax.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 = (javax.jms.ConnectionFactory)mdctx.lookup("targetCFRef");

      if (targetDest == null)
        targetDest = (javax.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) {
      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) {};
    }
  }
}
```

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

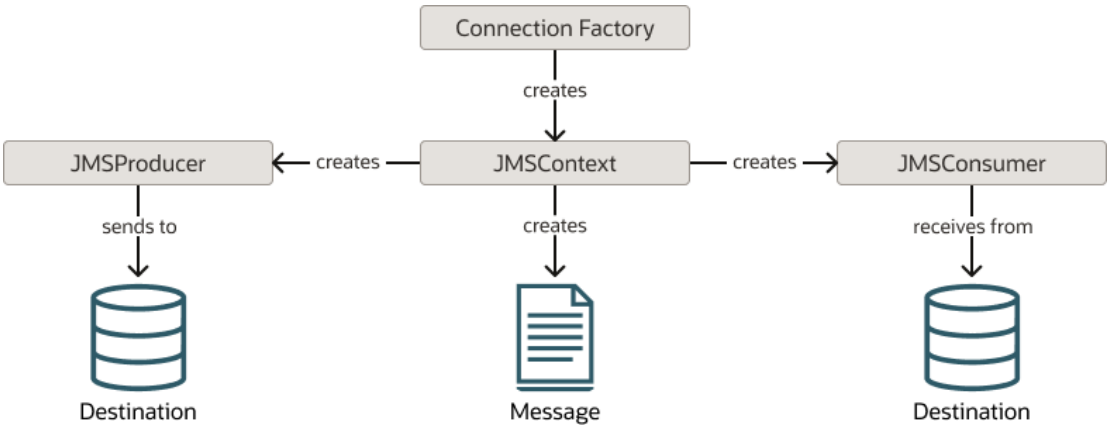
The JMS 2.0 simplified API provides the same basic functionality as the JMS 1.1 API (classic API), but the new interfaces and several API changes make it easier to use.

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

- `ConnectionFactory` : An administered object used by a client to create a `Connection`. This interface is also used by 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 about the JMS 2.0 interfaces, see the `javax.jms` package documentation at https://javaee.github.io/javaee-spec/javadocs/javax/jms/package-summary.html.

New Interfaces in the Simplified JMS API

The JMS 2.0 simplified API consists of three new interfaces.

- JMSContext
- JMSProducer
- JMSConsumer

JMSContext

The main interface in the simplified API is `JMSContext` . It combines the functions of the `Connection` and `Session` objects of the JMS 1.1 API. Creating a single `JMSContext` object eliminates the need to create a connection, session, and a text message separately.

For more information about the `JMSContext` interface, see https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSContext.html.

The `WLJMSContext` interface in the `weblogic.jms.extensions` package defines the fields and methods that are not supported by `javax.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 about the `JMSProducer` interface, see https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSProducer.html.

The `WLJMSProducer` interface defines methods and attributes specific to WebLogic JMS. You can use these features by casting the `JMSProducer` instance to the `WLSJMSProducer` interface defined in the `weblogic.jms.extensions` package. 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 about the `JMSConsumer` interface, see https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSConsumer.html.

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.

- Method to Extract the Body Directly from a Message
- Method to Receive a Message Body Directly
- Method to Create a Session

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.

```
void onMessage(Message message){ // delivers a BytesMessage
   byte[] bytes = message.getBody(byte[].class);
   ...
```

For more information, see the Javadoc at

https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html

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 Javadoc at:

https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSConsumer.html

Method to Create a Session

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

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
- Setting Up a JMS Application
- Sending Messages
- Receiving Messages
- Acknowledging Received Messages
- Releasing Object Resources

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 6-1 lists the packages that are commonly used by WebLogic JMS applications.

Table 5-1 WebLogic JMS Packages

Package	Description
`javax.jms`	JMS API. This package is always used by WebLogic JMS applications. See `https://javaee.github.io/javaee-spec/javadocs/javax/jms/package-summary.html`.
`javax.naming` weblogic.jndi	JNDI packages required for server and destination lookups. See `http://docs.oracle.com/javase/8/docs/api/javax/naming/package-summary.html`.
`javax.transaction.UserTransaction`	JTA API required for JTA user transaction support. See `http://www.oracle.com/technetwork/java/javaee/jta/index.html`.
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:

- Using a Simplified API to Set Up a JMS Application
- Using the Classic API to Set Up a 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.

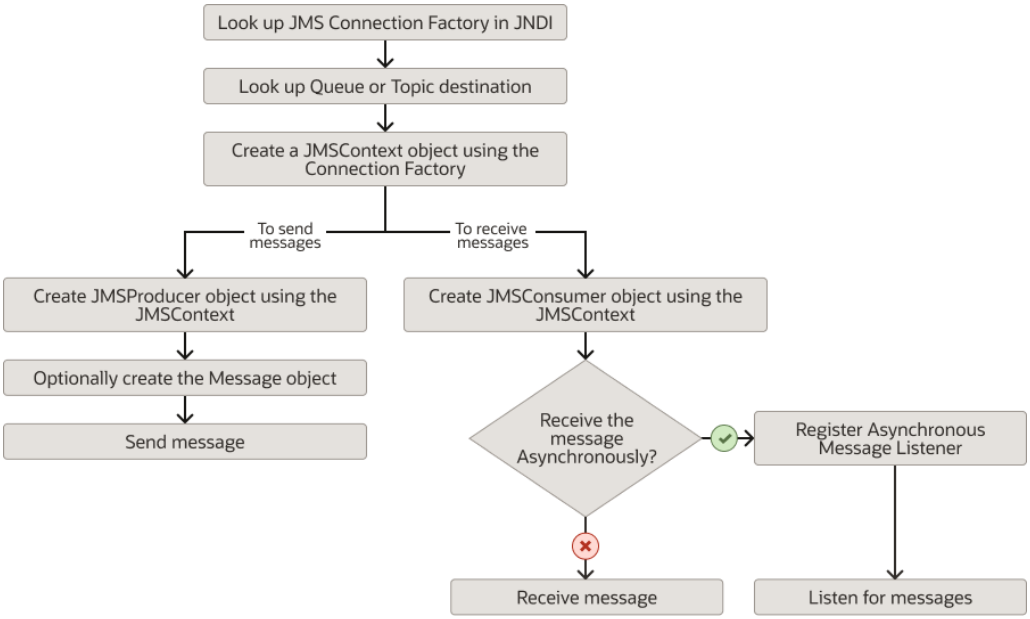
- For information about JMS resource definitions, see Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*.
- For information about configuring other JMS resources, see Configure Messaging in the *Oracle WebLogic Server Administration Console Online Help.*
- For more information about the JMS classes and methods described in these sections, see Understanding the JMS API, or the `javax.jms`, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/package-summary.html`, 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.

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.

The administrator can configure new connection factories during configuration; however, these factories must be uniquely named or the server will not boot. You can also use the default connection factories defined by the Java 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, you can look it up by establishing a JNDI context (`namingContext`) using the `InitialContext()` constructor, at `https://docs.oracle.com/javase/8/docs/api/javax/naming/InitialContext.html`. For any application other than a servlet application, you must provide a `Hashtable` defining the environment when calling the `InitialContext` constructor.

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);
```

The *`CF_name`* argument specifies the connection factory name defined during the configuration.

For more information about the `ConnectionFactory` class, see ConnectionFactory, or the `javax.jms.ConnectionFactory` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionFactory.html.

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 Configure topics and Configure queues in the *Oracle WebLogic Server Administration Console Online Help.* For more information, see Destination or the Javadocs at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Queue.html` and `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Topic.html`.

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 https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionFactory.html.

Alternatively, you can inject `JMSContext` in the Java 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 Java EE applications. For example:

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

For more information about using the `JMSContext` interface, see https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSContext.html.

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();
```

> **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 https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSProducer.html.

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 https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSConsumer.html.

Sending and Receiving Messages using the Simplified API

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

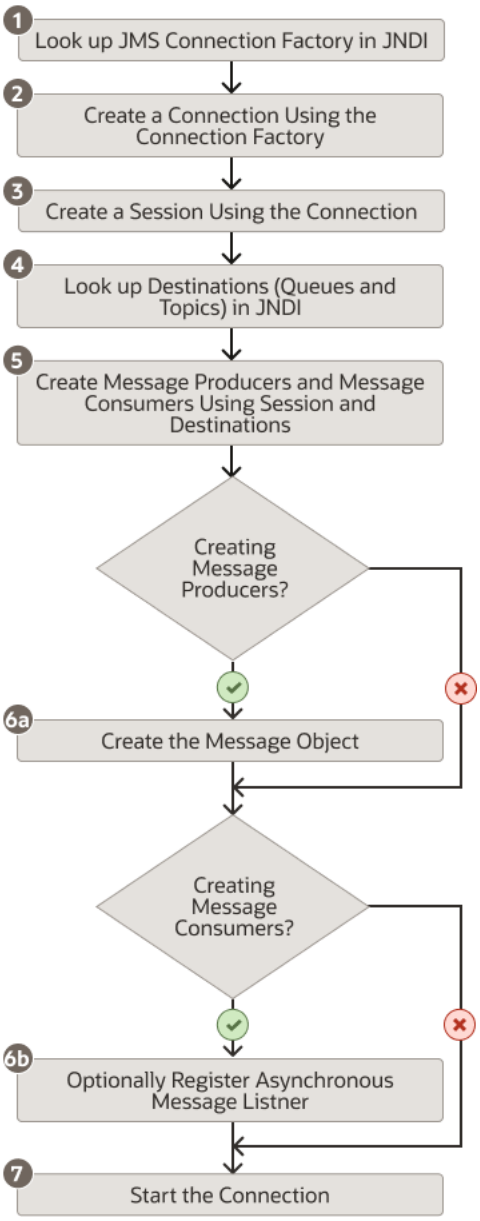
- Sending Messages Using the Simplified JMS API
- Sending a Message Asynchronously

- 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.

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.

The administrator can configure new connection factories during configuration; however, these factories must be uniquely named or the server will not boot. You can also use the default connection factories defined by the Java 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, you can look it up by establishing a JNDI context (`context`) using the `InitialContext()` method, at `http://docs.oracle.com/javase/8/docs/api/javax/naming/InitialContext.html#InitialContext()`. For any application other than a servlet application, you must pass an environment used to create the initial context.

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 name defined during configuration.

For more information about the `ConnectionFactory` class, see ConnectionFactory, or the `javax.jms.ConnectionFactory` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionFactory.html`.

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` class, see Connection, or the `javax.jms.Connection` Javadoc, at `http://docs.oracle.com/javaee/7/api/javax/jms/Connection.html`.

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` class methods, see the `javax.jms.QueueConnectionFactory` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionFactory.html. For more information about the `QueueConnection` class, see the `javax.jms.QueueConnection` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueConnection.html.

Create a Topic Connection

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

```
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` class methods, see the `javax.jms.TopicConnectionFactory` Javadoc., at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicConnectionFactory.html. For more information about the `TopicConnection` class, see the `javax.jms.TopicConnection` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicConnection.html.

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 session and its message producers and consumers can only be accessed by one thread at a time. Their behavior is undefined if multiple threads access them simultaneously.
>
> WebLogic JMS does not support having both types of MessageConsumer (QueueConsumer and TopicSubscriber) for a single Session. However, it does support a single session with both a QueueSender and a TopicSubscriber (and vice-versa: QueueConsumer and TopicPublisher), or with multiple MessageProducers of any type.

For more information about the `Session` class, see Session or the `javax.jms.Session` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Session.html`.

Create a Session Using the createSession Method

Use the `createSession` method in `javax.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,
 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` class methods, see the `javax.jms.QueueConnection` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueConnection.html`. For more information about the `QueueSession` class, see the `javax.jms.QueueSession` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueSession.html`.

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` class methods, see the `javax.jms.TopicConnection` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicConnection.html. For more information about the `TopicSession` class, see the `javax.jms.TopicSession` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicSession.html.

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 Configure topics and Configure queues in the *Oracle WebLogic Server Administration Console Online Help.* For more information about the `Destination` class, see Destination or the `javax.jms.Destination` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/Destination.html.

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/Subscibe 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.

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` classes, see MessageProducer and MessageConsumer, or the `javax.jms.MessageProducer`, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageProducer.html`, and `javax.jms.MessageConsumer` Javadocs, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageConsumer.html`.

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(
 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 the value of null to the `createSender()` method, you create an *anonymous producer*. In this case, you must specify the queue name when sending messages, as described in Sending Messages.

After the queue sender or receiver is created, you can access the queue name associated with the queue sender or receiver using the following `QueueSender` or `QueueReceiver` method:

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

For more information about the `QueueSession` class methods, see the `javax.jms.QueueSession` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueSession.html. For more information about the `QueueSender` and `QueueReceiver` classes, see the `javax.jms.QueueSender`, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueSender.html, and `javax.jms.QueueReceiver` Javadocs, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueReceiver.html.

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 methods described in this section create non-durable subscribers. Non-durable topic subscribers only receive messages sent while they are active. For information about the methods used to create durable subscriptions enabling messages to be retained until all messages are delivered to a durable subscriber, see Creating Subscribers for a Durable Subscription. In this case, durable subscribers only receive messages that are published after the subscriber has subscribed.

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 a value of null to the `createPublisher()` method, then you create an *anonymous producer*. In this case, you must specify the topic name when sending messages, as described in Sending Messages.

An application can have JMS connections that it uses to both publish and subscribe to the same topic. Because topic messages are delivered to all subscribers, the application can receive messages it has published itself. To prevent this behavior, a JMS application can set a `noLocal` flag to `true`.

After the topic publisher or subscriber is created, you can access the topic name associated with the topic publisher or subscriber using the following `TopicPublisher` or `TopicSubscriber` method:

```
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` class methods, see the `javax.jms.TopicSession` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicSession.html. For more information about the `TopicPublisher` and `TopicSubscriber` classes, see the `javax.jms.TopicPublisher`, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicPublisher.html, and the `javax.jms.TopicSubscriber` Javadocs, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicSubscriber.html.

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

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` class methods, see the `javax.jms.Session`, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Session.html`, and `weblogic.jms.extensions.WLSession` Javadocs, respectively. For more information about the `Message` class and its methods, see Messages, or the `javax.jms.Message` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html`.

Step 6b: Optionally Register an Asynchronous Message Listener

> **Note:**
>
> This step applies to message consumers only.

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

1. Implement the `javax.jms.MessageListener` interface, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageListener.html`, 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.
>
> If you want to issue the `close()` method within an `onMessage()` method call, the system administrator must select the Allow Close In OnMessage option when configuring the connection factory. For more information on configuring connection factory options, see "Configuring Basic JMS System Resources" in *Administering JMS Resources for Oracle WebLogic Server*.

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.

You can unset a message listener by calling the `MessageListener()` method with the value of null.

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` class methods are inherited by the `QueueReceiver` and `TopicSubscriber` classes. For additional information about the `MessageConsumer` class methods, see MessageProducer and MessageConsumer or the `javax.jms.MessageConsumer` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageConsumer.html`.

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 `javax.jms.Connection` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Connection.html`.

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 in the *EXAMPLES_HOME*`\wlserver\samples\server\examples\src\examples\jms\classicapi\queue` directory where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured. 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
        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.

```
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 session, it is passed to the `examples.jms.QueueReceive.onMessage()` method. The following code example shows the `onMessage()` interface from the `QueueReceive` example:

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

                  quit = true;
                this.notifyAll(); // Notify main thread to quit
          }
        }
    } catch (JMSException jmse) {
     jmse.printStackTrace();
    }
}
```

The `onMessage()` method processes messages received through the queue receiver. The method verifies that the message is a `TextMessage` and, if it is, prints the text of the message. If the `onMessage()` method receives a different message type, then it uses the message's `toString()` method to display the message contents.

> **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 `javax.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 in the *EXAMPLES_HOME*`\wlserver\samples\server\examples\src\examples\jms\classicapi\topic` directory, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured. 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
{
  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 queue. 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.

```
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 session, it is passed to the `examples.jms.TopicSubscribe.onMessage()` method. The `onMessage()` interface for the `TopicReceive` example is the same as the `QueueReceive onMessage()` interface, as described in Example: Setting Up a Point-to-Point JMS Application Using the Classic API.

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

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,

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

This code creates a `TextMessage` object and sets its body to "Hello", and then sends it to the specified `destination`.

You can also use the following `JMSProducer` methods, which create a message automatically for of the appropriate message type and set the payload to the specified parameter:

```
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:

https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSProducer.html

WebLogic JMS provides proprietary attributes that you can use while sending messages. You can specify the delivery mode (`DeliveryMode.PERSISTENT` or `DeliveryMode.NON_PERSISTENT`), priority (`0-9`), delivery delay, and time-to-live (in milliseconds) by casting the `JMSProducer` instance to `weblogic.jms.extensions.WLJMSProducer`. See the Javadoc for `WLSJMSProducer` in *Java API Reference for Oracle WebLogic Server*.

For example,

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

If not specified, the delivery mode, priority, and time-to-live attributes are set to one of the following:

- Connection factory or destination override configuration attributes defined for the producer, as described Configure default delivery parameters in the *Oracle WebLogic Server Administration Console Online Help*.
- Values specified using the JMSProducer object's set methods, as described in Setting JMSProducer and MessageProducer Attributes.

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:

1. Create a Message Object
2. Define a Message
3. Send the Message to a Destination Using MessageProducer

For more information about the JMS classes for sending messages and the message types, see the `javax.jms.Message` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html`. 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 a value was specified and you do not want to change it, you can go to step 3.

If a value was specified or if you want to change an existing value, you can define a value using the appropriate `set` method. For example, the method for defining the text of a `TextMessage` is as follows:

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

> **Note:**
>
> Messages can be defined as 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 `javax.jms.Session` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Session.html`.

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:**
>
> If multiple topic subscribers are defined for the same topic, each subscriber will receive its own local copy of a message. 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 the write permission.

For more information about the `MessageProducer` class, see MessageProducer and MessageConsumer or the `javax.jms.MessageProducer` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageProducer.html`.

You must specify a message. You can also specify the queue name (for anonymous message producers), delivery mode (`DeliveryMode.PERSISTENT` or `DeliveryMode.NON_PERSISTENT`), priority (`0-9`), delivery delay, and time-to-live (in milliseconds). If not specified, the delivery mode, priority, and time-to-live attributes are set to one of the following:

- Connection factory or destination override configuration attributes defined for the producer, as described in Configure default delivery parameters in the *Oracle WebLogic Server Administration Console Online Help*.
- Values specified using the message producer's set methods, as described in Setting JMSProducer and MessageProducer Attributes.

If you define the delivery mode as `PERSISTENT`, you should configure a backing store for the destination, as described in Configure persistent stores in the *Oracle WebLogic Server Administration Console Online Help*.

> **Note:**
>
> If no backing store is configured, then the delivery mode is changed to `NON_PERSISTENT` and messages are not written to the persistent store.

For more information about using the `QueueSender` and `TopicPublisher` methods for sending messages, see the WebLogic Server documentation at:

`https://docs.oracle.com/middleware/1213/wls/JMSPG/implement.htm#JMSPG228`

For additional information about the `QueueSender` class methods, see the `javax.jms.QueueSender` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueSender.html`.

For more information about the `TopicPublisher` class methods, see the `javax.jms.TopicPublisher` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicPublisher.html.

Sending a Message Asynchronously

In asynchronous mode, the JMS client sends a message and returns control to the application without waiting for an acknowledgement from the JMS server.

To send messages asynchronously, your application should define a `CompletionListener` object. When an acknowledgement is received from the JMS server to indicate that the message was received, the JMS provider notifies the application by invoking the callback method `onCompletion` on the `CompletionListener` object defined by the application. For more information about the `CompletionListener` interface, see https://javaee.github.io/javaee-spec/javadocs/javax/jms/CompletionListener.html.

After defining the `javax.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);
  ```

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

Setting JMSProducer and MessageProducer Attributes

As described in the previous section, when sending a message, you can optionally specify the delivery mode, priority, delivery delay, and time-to-live values. If not specified, these attributes are set to the connection factory configuration attributes, as described in Configure connection factories in the *Oracle WebLogic Server Administration Console Online Help*.

Alternatively, you can set the delivery mode, priority, time-to-deliver, time-to-live, and redelivery delay (timeout), and redelivery limit values dynamically using the message producer's set methods. Table 6-2 lists the message producer set and get methods for each dynamically configurable attribute.

> **Note:**
>
> The delivery mode, priority, time-to-live, time-to-deliver, redelivery delay (timeout), and redelivery limit attribute settings can be overridden by the destination using the Delivery Mode Override, Priority Override, Time To Live Override, Time To Deliver Override, Redelivery Delay Override, and Redelivery Limit configuration attributes, as described in Configure message delivery overrides and Configure topic message delivery overrides in the *Oracle WebLogic Server Administration Console Online Help*.

Table 5-2 Message Producer Set and Get Methods

Attribute	Set Method	Get Method
Delivery Mode	`public void setDeliveryMode( int deliveryMode ) throws JMSException`	`public int getDeliveryMode( ) throws JMSException`
Priority	`public void setPriority( int defaultPriority ) throws JMSException`	`public int getPriority( ) throws JMSException`
Time-to-Live	`public void setTimeToLive( long timeToLive ) throws JMSException`	`public long getTimeToLive( ) throws JMSException`
Redelivery limit	`public void setRedeliveryLimit( int redeliveryLimit ) throws JMSException`	`public int getredeliveryLimit( ) throws JMSException`
Send timeout	`public void setsendTimeout( long sendTimeout ) throws JMSException`	`public long getsendTimeout( ) throws JMSException`

> **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` class methods, see the `javax.jms.MessageProducer` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageProducer.html`, or the `weblogic.jms.extensions.WLMessageProducer` Javadoc.

Example: Sending Messages Within a Point-toPoint Application

The following example is excerpted from the `examples.jms.queue.QueueSend` 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. 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` class and methods, see the `javax.jms.QueueSender` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueSender.html.

Example: Sending Messages Within a Publish/Subscribe Application

The following example is excerpted from the `examples.jms.topic.TopicSend` example, provided with WebLogic Server in the *EXAMPLES_HOME*`\wl_server\examples\src\examples\jms\topic` directory, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured. 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` class and methods, see the `javax.jms.TopicPublisher` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicPublisher.html.

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 destination keys in the *Oracle WebLogic Server Administration Console Online Help*.

After the message 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 classes for receiving messages and the message types, see the `javax.jms.Message` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html`. For information about sending messages, see Sending Messages.

Receive Messages Asynchronously Using the Simplified API

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. Message delivery is started automatically.

```
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.

Asynchronous Message Pipeline

If messages are produced faster than asynchronous message listeners (consumers) can consume them, a JMS server will push multiple unconsumed messages in a batch

to another session with available asynchronous message listeners. These in-flight messages are sometimes referred to as the *message pipeline*, or in some JMS vendors as the *message backlog*. The pipeline or backlog size is the number of messages that are accumulated on an asynchronous consumer, but that are not been passed to a message listener.

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 connection factories in the *Oracle WebLogic Server Administration 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 are sometimes aggregated into a single message on the network transport. If the messages are sufficiently large, the aggregate size of the data written may exceed the maximum value for the transport, which may cause undesirable behavior. For example, the `t3` protocol sets a default maximum message size of 10,000,000 bytes, and is configurable on the server with the `MaxT3MessageSize` attribute. This means that if ten 2 megabyte messages are in the pipeline `t3` limit may be exceeded.

Receive Messages Synchronously Using the Simplified API

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

```
public String receiveMessage(
ConnectionFactory connectionFactory,Queue queue){
```

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

For additional information about the `JMSConsumer` class methods, see the `javax.jms.JMSConsumer` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/JMSConsumer.html.

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
```

In each case, the application receives the next message produced. If you call the `receive()` method with no arguments, then the call blocks indefinitely until a message is produced or the application is closed. Alternatively, you can pass a timeout value to specify how long to wait for a message. If you call the `receive()` method with a value of 0, then the call blocks indefinitely. The `receiveNoWait()` method receives the next message if one is available, or returns null; in this case, the call does not block.

The `MessageConsumer` class methods are inherited by the `QueueReceiver` and `TopicSubscriber` classes. For additional information about the `MessageConsumer` class methods, see the `javax.jms.MessageConsumer` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageConsumer.html.

Example: Receiving Messages Synchronously Within a PTP Application

The following example is excerpted from the `examples.jms.queue.QueueReceive` 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. 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 in the *EXAMPLES_HOME*`\wl_server\examples\src\examples\jms\topic` directory, where *EXAMPLES_HOME* represents the directory in which the WebLogic Server code examples are configured. 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

In releases prior to WebLogic Server 9.1, synchronous consumers required making a two-way network calls for each message, which was an inefficient model because the synchronous consumer could not retrieve multiple messages, and could also increase network traffic resources, since synchronous consumers would continually poll the server for available messages. In WebLogic 9.1 or later, 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, either using the WebLogic Server Administration Console or the `JMSClientParamsBean` MBean.

Similar to the asynchronous message pipeline, when the Prefetch Mode is enabled on a JMS client's connection factory, the connection factory's targeted JMS servers will proactively push batches of unconsumed messages to synchronous message consumers, using the connection factory's Messages Maximum per Session parameter to define the maximum number of messages per batch. This may improve performance because messages are ready and waiting for synchronous consumers when the consumers are ready to process more messages, and it may also reduce network traffic by reducing synchronous calls from consumers that must otherwise continually poll for messages.

Synchronous message prefetching does not support user (XA) transactions for synchronous message receives or multiple synchronous consumers per session (regardless of queue or topic). In most such cases, WebLogic JMS will silently and safely ignore the Prefetch Mode for Synchronous Consumer flag; however, otherwise WebLogic will fail the application's synchronous receive calls.

For more information on the behavior of pipelined messages, see Asynchronous Message Pipeline. For more information on configuring a JMS connection factory, see Configure connection factories in the *Oracle WebLogic Server Administration Console Online Help*.

Recovering Received Messages

> **Note:**
>
> This section applies only to non-transacted sessions for which the acknowledge mode is set to `CLIENT_ACKNOWLEDGE`. Synchronously received `AUTO_ACKNOWLEDGE` messages may not be recovered; they have already been acknowledged.

An application can request that JMS redeliver messages (unacknowledge them) using the following method:

```
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
- 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

When you finish using the connection, session, message producer or consumer, connection consumer, or queue browser created on behalf of a JMS application, you should explicitly close them to release the resources.

Enter the `close()` method to close JMS objects, as follows:

```
public void close(
) throws JMSException
```

When closing an object:

- The call blocks until the method call completes or until any outstanding asynchronous receiver `onMessage()` calls complete.
- 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 `javax.jms` Javadoc, at `http://www.oracle.com/technetwork/java/jms/index.html`. 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.QueueSend` example, provided with WebLogic Server in the *`EXAMPLES_HOME`*`\wl_server\examples\src\examples\jms\queue` directory. *`EXAMPLES_HOME`* represents the directory in which the WebLogic Server code examples are configured. 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 `QueueSend` example, the `close()` method is called at the end of `main()` to close objects and free resources.

6

Managing Your Applications

Learn how to manage your JMS applications programatically using the value-added WebLogic JMS features.

Managing Rolled Back, Recovered, Redelivered, or Expired Messages

Learn how to mange 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. The `RedeliveryDelay` attribute of a connection factory is configured using the WebLogic Server Administration Console.

For more information, see Configure connection factories in the *Oracle WebLogic Server Administration Console Online Help*.

The application that creates the session can then override the connection factory setting using WebLogic-specific extensions to the `javax.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.

> **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 `javax.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.

The `RedeliveryDelayOverride` attribute of a destination is configured using the WebLogic Server Administration Console. For more information, see:

- Configure queue message delivery failure options in the *Oracle WebLogic Server Administration Console Online Help*

- Configure topic message delivery failure options in the *Oracle WebLogic Server Administration Console Online Help*

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.

Configuring a Message Redelivery Limit on a Destination

When a destination's attempts to redeliver a message to a consumer reaches a specified redelivery limit, then the destination deems the message undeliverable. The `RedeliveryLimit` attribute is set on a destination and is configurable using the WebLogic Server Administration Console. This setting overrides the redelivery limit set on the message producer. For more information, see:

- Configure queue message delivery failure options in the *Oracle WebLogic Server Administration Console Online Help*.
- Configure topic message delivery failure options in the *Oracle WebLogic Server Administration Console Online Help*.

Configuring an Error Destination for Undelivered Messages

If an error destination is configured on the JMS server for undelivered messages, then when a message has been deemed undeliverable, the message will be redirected to a specified error destination. The error destination can be either a queue or a topic, and it must be configured on the same JMS server as the destination for which it is defined. If no error destination is configured, then undeliverable messages are simply deleted.

The `ErrorDestination` attribute is configured for standalone destinations and uniform distributed destination using the WebLogic Server Administration Console. For more information, see:

- Configure queue message delivery failure options in the *Oracle WebLogic Server Administration Console Online Help*.
- Configure topic message delivery failure options in the *Oracle WebLogic Server Administration Console Online Help*.
- Uniform distributed queues - configure delivery failure parameters in the *Oracle WebLogic Server Administration Console Online Help*.
- Uniform distributed topics - configure delivery failure parameters in the *Oracle WebLogic Server Administration Console Online Help*.

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*.

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 Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

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 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 `javax.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.

> **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 `javax.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 Relative Time-to-Deliver Override

The relative `TimeToDeliverOverride` attribute is a string specified as an integer, and is configurable using the WebLogic Server Administration Console.

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
`* * * 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. 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 `WLSession` 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:

```
public static long nextScheduledTimeInMillisRelative(
  String schedule
 ) throws ParseException {
```

Managing Connections

Learn how to manage JMS connections.

- Defining a Connection Exception Listener
- Accessing Connection Metadata
- Starting, Stopping, and Closing a Connection

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.

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`

Table 6-2 (Cont.) JMS Metadata

JMS Metadata	Get Method
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 `javax.jms.ConnectionMetaData` Javadoc at https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionMetaData.html.

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.

- 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
- Closing a Session

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
```

> **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.

- Dynamically Creating Destinations
- Dynamically Deleting Destinations

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)
- Administration console
- 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. For more information, see Configuring an Error Destination for Undelivered Messages.

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 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.

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 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.

> **Note:**
>
> Temporary destinations are enabled by default through the JMS server's `Hosting Temporary Template` attribute. However, if you want to create temporary destinations with specific settings, you must modify the default `Temporary Template` values using the JMS server's `Temporary Template` and `Module Containing Temporary Template` attributes, as explained in Configure general JMS server properties in the *Oracle WebLogic Server Administration Console Online Help*.

The following sections describe how to create a temporary queue (Point-to-Point) or temporary topic (Publish/Subscribe).

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();
```

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
- Setting the Client ID Policy
- Defining the Client ID
- Creating a Sharable Subscription Policy
- Creating Subscribers for a Durable Subscription
- Best Practice: Always Close Failed JMS ClientIDs
- Deleting Durable Subscriptions
- Modifying Durable Subscriptions
- Managing Durable Subscriptions

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:

- `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*.

To set the `Client ID Policy` attribute on the connection factory using the WebLogic Console, see Configure multiple connections using the same client Id in the *Oracle WebLogic Server Administration Console Online Help*. The connection factory setting can be overridden programatically using the `setClientID` method of the `WLConnection` interface in *Java API 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. See in *Oracle WebLogic Server Administration Console Online Help*.
- 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.

> **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/Subscibe 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 `javax.jms.JMSContext` instance to `weblogic.jms.extensions.WLJMSContext` or a `javax.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.

For more information on how to use the Subscription Sharing policy, see:

- Configure a connection factory subscription sharing policy in *Oracle WebLogic Server Administration Console Online Help*.
- 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.

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.

> **Note:**
>
> You can also delete durable subscriptions from the WebLogic Server Administration Console. 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.

   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 using either the WebLogic Server Administration Console or 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 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 7-3 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
JMSDeliveryMode1	public void setJMSDeliveryMode(int deliveryMode) throws JMSException	public int getJMSDeliveryMode() 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
JMSReplyTo	public void setJMSReplyTo(Destination replyTo) throws JMSException	public Destination getJMSReplyTo() throws JMSException
JMSTimeStamp1	public void setJMSTimeStamp(long timestamp) throws JMSException	public long getJMSTimeStamp() throws JMSException

Table 6-3 (Cont.) JMS Header Field Methods

Header Field	Set Method	Get Method
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.

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, int, long, short, and string.`

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 7-4, 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.

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**
Object	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**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 `javax.jms.Message` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html`.

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
```

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 `javax.jms.QueueBrowser` Javadoc at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueBrowser.html`.

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);
```

For more information about the message selector syntax, see the `javax.jms.Message` Javadoc at https://javaee.github.io/javaee-spec/javadocs/javax/jms/Message.html.

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'";
```

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(
  (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

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:

1. Serialize the XML to a `StringWriter`.
2. Call the`toString` on the `StringWriter` and pass it into the `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.

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*.

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
- Deleting a JMS System Resource

Creating a JMS System Resource

The module contains a connection factory and a topic.

Example 8-1 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());

//
// 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 8-2 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();
     }
}
.
.
.
```

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 allow 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.

- Trap for null MBean objects (such as servers, JMS servers, modules) before trying to manipulate the MBean object.
- 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, and if unsuccessful, sleeps for a specified interval before retrying. A maximum retry count is established to prevent an infinite loop.

```
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();
  for (int i=retryCount; i>=0; i--) {
       try {
        System.out.println("Trying " + command);
        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 (retryCount == 0) throw je;
       }
       try {
        System.out.println(command + "> failed, pausing " +
               retryInterval + " millis.");
        Thread.sleep(retryInterval);
       } catch (InterruptedException ignore) {}
  }
  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);
```

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.

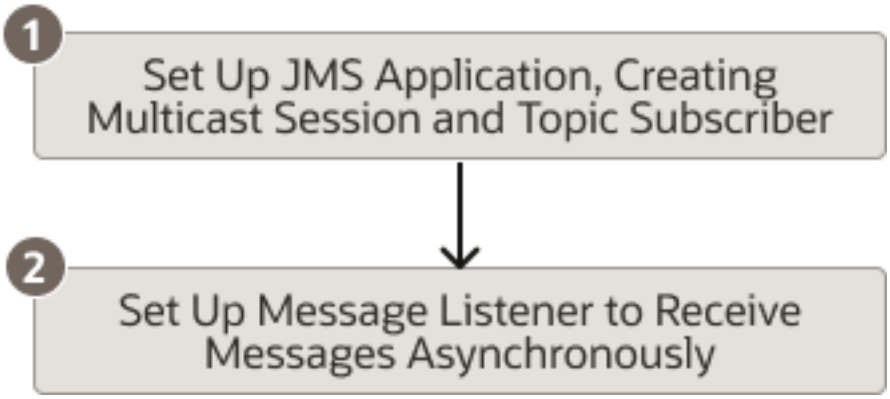
See Communications In a Cluster in *Administering Clusters for Oracle WebLogic Server*.

Configuring Multicasting for WebLogic Server

Learn how to configure multicasting for WebLogic server.

Figure 9-1 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.

For more information, see Configure topic multicast parameters in the *Oracle WebLogic Server Administration Console Online Help*.

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/Subscibe 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/Subscibe messaging model.

```
JMSModuleHelper.createPermanentQueueAsync(ctx, domain, jmsServerName,
 queueName, jndiName);
Queue queue = findQueue(qsess, jmsServerName, queueName,
 retry_count, retry_interval);
```

> **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);
```

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 9-2, 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.

If the message maximum is reached, a `DataOverrunException` is thrown and messages are automatically discarded based on the overrun policy. Alternatively, you can set the messages maximum and overrun policy using the `Session` set methods.

Table 9-1 lists the `Session` set and `get methods` for each dynamically configurable attribute.

Table 8-1 Message Producer 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. For more information about these multicast configuration attributes, see Configure topic multicast parameters in the *Oracle WebLogic Server Administration Console Online Help*.

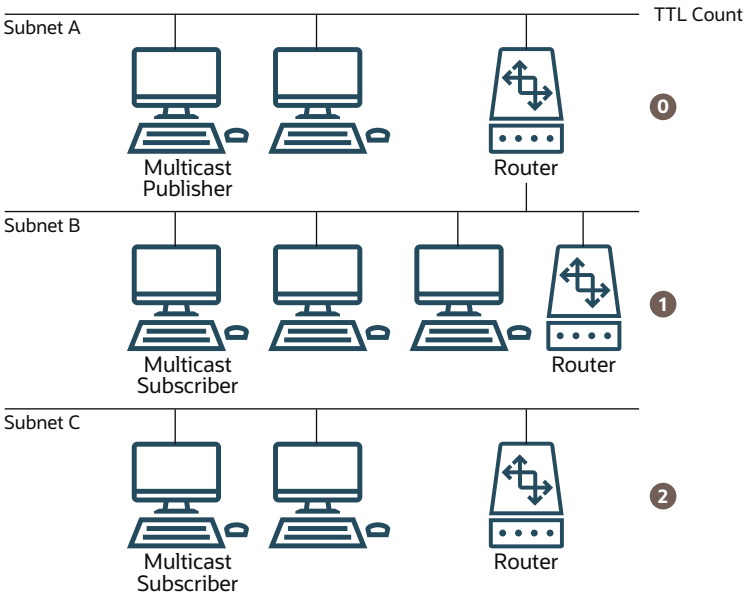
Example: Multicast Time-to-Live

> **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 9-1 shows how the Multicast TTL destination configuration attribute affects the delivery of messages across routers.

For more information, see Configure topic multicast parameters in the *Oracle WebLogic Server Administration Console Online Help*.

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.

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 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

System administrators create distributed destinations by using the WebLogic Server Administration Console.

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.

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 can not 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 `javax.jms.Destination` interface, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/Destination.html`, 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
- Using Replicated Distributed Topics
- Using Partitioned Distributed Topics

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. '

This section provides information on using distributed queues:

- Queue Forwarding
- QueueSenders
- QueueReceivers
- QueueBrowsers

Queue Forwarding

Queue members can forward messages to other queue members by configuring the `Forward Delay` attribute in the WebLogic Server Administration Console, which is disabled by default. This attribute defines the amount of time, in seconds, that a distributed queue member with messages, but which has no consumers, will wait before forwarding its messages to other queue members that do have consumers. By default, WebLogic Server resets the delivery count when forwarding between distributed queue members. See `Reset Delivery Count On Forward`.

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 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
- TopicSubscribers
- Deploying Message-Driven Beans on a Distributed Topic

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 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.

> **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 no -durable, 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 .

> **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 on which a queue producer is created fails, yet the WebLogic Server instance where the producer's JMS connection resides is still running, then the producer remains active and WebLogic JMS will fail it over to another distributed queue member, irrespective of whether the Load Balancing option is enabled. For example, a WebLogic cluster contains WLSServer1, WLSServer2, and WLSServer3 and you are connected to WLServer2. If server WLSServer 2 fails, WebLogic JMS fail the producer over to one of the remaining cluster members. For more information, see Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*.

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.

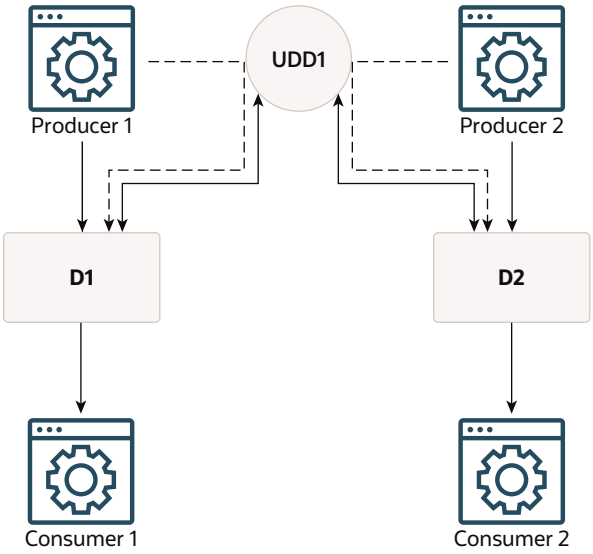
The following sections provide common use case scenarios when using distributed destinations:

- Maximizing Production
- Maximizing Availability
- Stuck Messages

Maximizing Production

To maximize message production, Oracle recommends that each member of a distributed destination be associated with a producer and a consumer. Figure 10-1 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

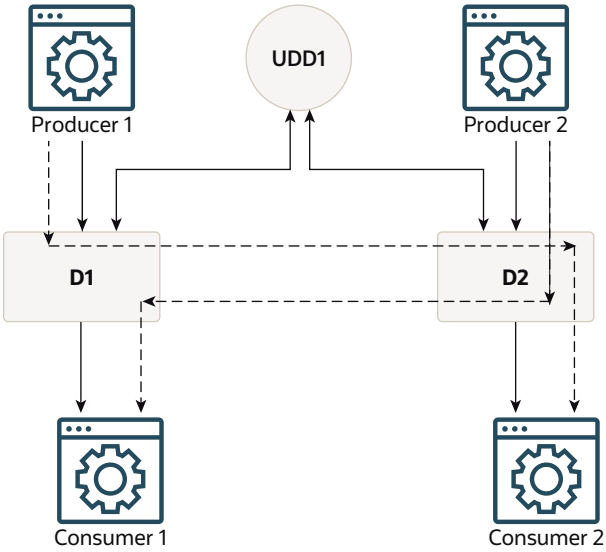
Ideally, its best to pair a producer with a consumer but it is not always practical. The rate that messages are consumed is the limiting factor that determines the message throughput of your application. You can increase the availability of consumers by using load balancing between member destinations. 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 under utilize resources.

Using Topics

When using a distributed topic, every member destination will forward its messages to every other member of the distributed topic.

Figure 9-2 Using Distributed Topics

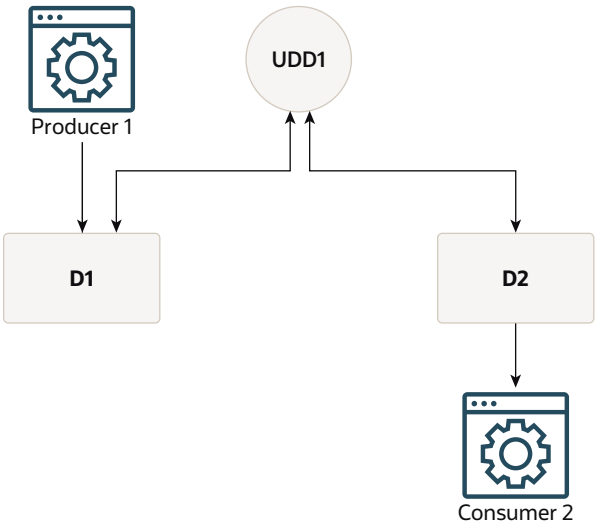


In Figure 10-2, UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. Each physical destination has a producer/consumer pair. Each consumer receives messages sent by Producer 1 and Producer 2.

Stuck Messages

InFigure 10-3, 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.

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** and requires that all messages from that unit be processed sequentially in the order they were created.

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 Java Message Service Specification, at `http://www.oracle.com/technetwork/java/jms/index.html`, 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 processed sequentially in the order the messages were created. 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 `http://www.oracle.com/technetwork/java/jms/index.html`, when using Message Unit-of-Order:

- 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 were created. 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.
- A queue that has several messages from the same Unit-of-Order must complete processing all of them before they can be delivered to any queue consumer or the next message can be delivered to the queue.

  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
- What Happened to Joe's Order
- How Message Unit-of-Order Solves the Problem

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.

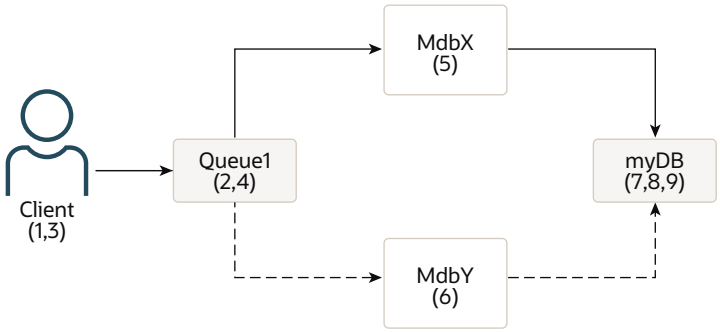
Joe logs in his XYZ Online Bookstore account and searches his favorite book topics. He chooses a book, proceeds to the checkout, and completes the sales transaction. Then Joe realizes he has previously purchased this same item, so he cancels the order. One week later, the book is delivered to Joe.

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 11-1 and the corresponding actions demonstrate how Joe's order was processed.

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.

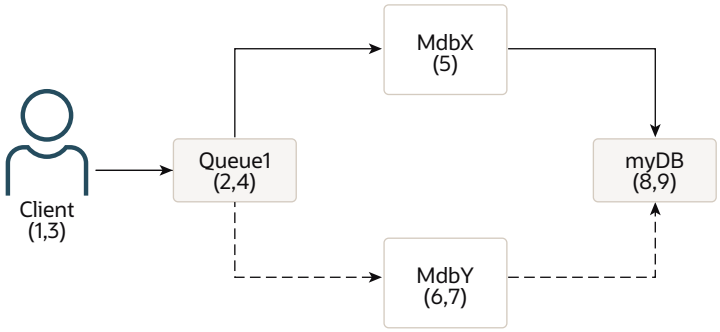
Although the Java Message Service Specification, at `http://www.oracle.com/technetwork/java/jms/index.html`, provides an ordered message delivery, it only provides ordered message delivery between a single instance of a producer and a single instance of a consumer. In Joe's case, multiple MDBs were available 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. All messages created during Joe's user session are processed sequentially in the order they were created because WebLogic Server guarantees that messages in a Unit-of-Order are not processed in parallel.

In Figure 11-2 and the corresponding actions demonstrate how Joe's order was processed using Message Unit-of-Order.

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.

- Creating a Unit-of-Order Programmatically
- Creating a Unit-of-Order Administratively
- Unit-of-Order Naming Rules

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.

The Example 11-1 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);
.
.
.
```

Creating a Unit-of-Order Administratively

The following section provides information about how to configure JMS connection factories or JMS destinations to enable Message Unit-of-Order.

Configuring Unit-of-Order for a Connection Factory and Destinations

Use one of the following methods to configure JMS connection factories and destinations to enable Message Unit-of-Order:

- Configure a connection factory to always use a user-generated Unit-of-Order name. As a result, all producers created from such a connection factory have Unit-of-Order enabled. See Configure connection factory unit-of-order parameters in the *Oracle WebLogic Server Administration Console Online Help*.
- Configure a connection factory to always use a system-generated Unit-of-Order name for each session. See Configure connection factory unit-of-order parameters in the *Oracle WebLogic Server Administration Console Online Help*.
- A client can call `WLProducer.setUnitOfOrder(name)` and change the initial connection factory setting on the producer.
- Configure a standalone or distributed destination to always use a system-generated Unit-of-Order name. See the following topics in the *Administration Console Online Help*:
  - Configure advanced topic parameters

- Configure advanced queue parameters
- Uniform distributed topics - configure advanced parameters
- Uniform distributed queues - configure advanced parameters
- Configure advanced JMS template parameters

You should administratively configure a Unit-of-Order on a connection factory or destination when interoperating with legacy JMS applications. This method provides a simple mechanism to ensure messages are processed in the order they are created without making any code changes.

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");
```

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 and in the same Unit-of-Order on a queue that is sorted by priority and the sort order is depending, 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*.

Using Unit-of-Order with Distributed Destinations

As previously discussed in the Message Processing According to the JMS Specification, the Java Message Service Specification (at `http://www.oracle.com/technetwork/java/jms/index.html`) 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
- Using Hash-Based Routing

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 can not meet the quality-of-service required ; only one distributed destination member consumes messages for a particular Unit-of-Order.

Configuring Routing on Uniform Distributed Destinations

See the following topics to configure either the Path service or hash-based routing mechanism on uniform distributed destinations using Message Unit-of-Order:

- Uniform distributed topics - configure advanced parameters in the *Oracle WebLogic Server Administration Console Online Help*
- Uniform distributed queues - configure advanced parameters in the *Oracle WebLogic Server Administration Console Online Help*

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.

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*.

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 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 under utilize resources.
- This release of WebLogic Server Message Unit-of-Order does not support clients connecting to a non-Unit-of-Order JMS provider (releases before than WebLogic 9.0 or non-WebLogic JMS providers).

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

Understand the basic UOW terminology and the rules for processing UOW messages.

- Message Unit-of-Work Case Study

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.
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. If 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.

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 re route 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 non book items, 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

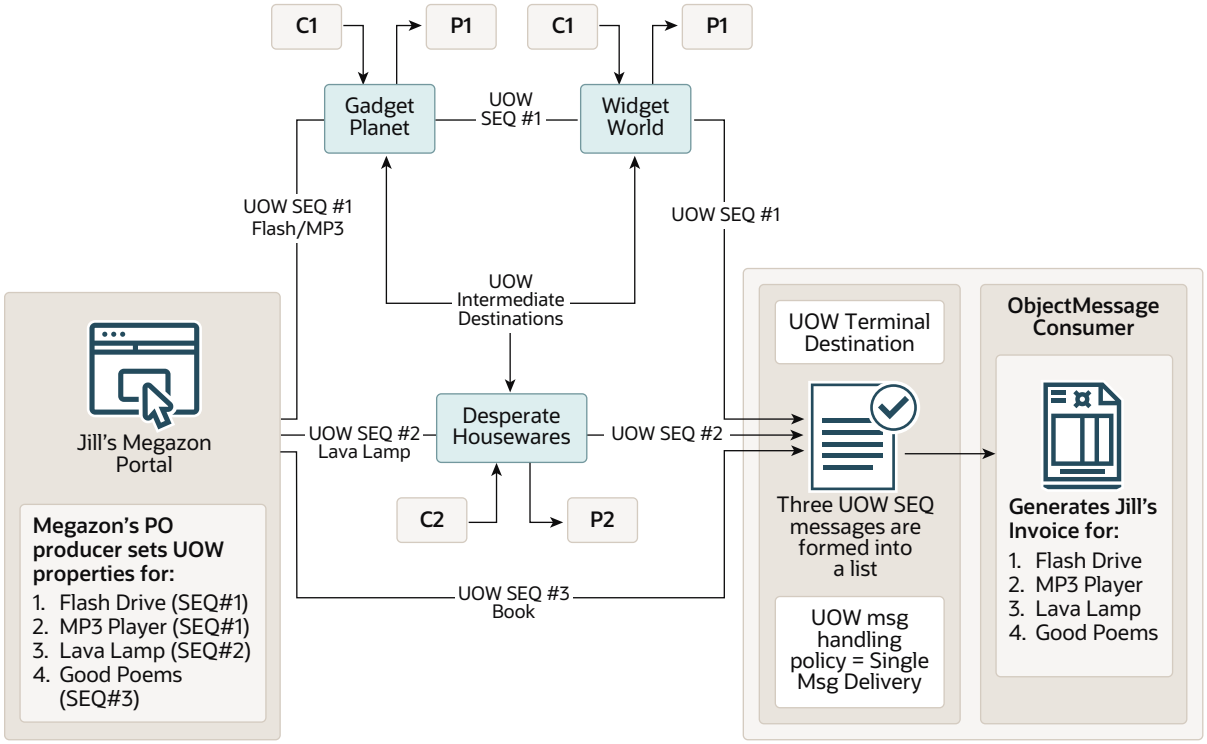
Jill logs into her Megazon account and does some holiday shopping. She chooses a book, flash drive, MP3 player, and a lava lamp, she then proceeds to the checkout, and completes the sales transaction.

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. Last, the system administrator for Megazon configures the terminal destination to UOW Message Handling Policy parameter to Single Message Delivery. See How to Create a Unit-of-Work Message Group.

Figure 12-1 and the corresponding actions demonstrate how Jill's order was processed using Message Unit-of-Work.

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#1 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.

- How to Write a Producer to Set UOW Message Properties
- How to Write a UOW Consumer/Producer For an Intermediate Destination

- Configuring Terminal Destinations
- How to Write a UOW Consumer for a Terminal Destination

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 11-1 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
`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 12-2copies the UOW properties defined in Table 11-1.

Example 11-1 Sample UOW Producer Message Properties

```
for (int i=1; i<=100; i++)
{
  sendMsg.setStringProperty("JMS_BEA_UnitOfWork","joule");
  sendMsg.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber",i);
  if (i == 100)
  {
```

```
        System.out.println("set the end of message flag for message # " + i);
        sendMsg.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd",true);
      }
      qSender.send(sendMsg, DeliveryMode.PERSISTENT,7,0);
    }
```

UOW Exceptions

The following exceptions may be thrown to the producer when sending JMS messages to a terminal destination. When a UOW exception is town, the UOW message is not delivered.

Except for the last one, they are all in the `weblogic.jms.extensions` package and are subclasses of `JMSException`.

- `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.
- `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 11-1.

Example 11-2 Sample Client Code for UOW Intermediate Destination

```
msg = qReceiver1.receive();
try
{
  text = msg.getText();
  TextMessage forwardmsg = qsess.createTextMessage();
  forwardmsg.setText(text);
  forwardmsg.setStringProperty("JMS_BEA_UnitOfWork",
              msg.getStringProperty("JMS_BEA_UnitOfWork"));
  forwardmsg.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber",
                 msg.getIntProperty("JMS_BEA_UnitOfWorkSequenceNumber"));
  if (tm.getBooleanProperty("JMS_BEA_IsUnitOfWorkEnd"))
    forwardmsg.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd",
              msg.getBooleanProperty("JMS_BEA_IsUnitOfWorkEnd"));
  qsend.send(forwardmsg);
}
```

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.

Using the WebLogic Server Administration Console, these Advanced configuration options are available on the General Configuration page for all destination types (or by using the `DestinationBean` API), as well as on JMS templates (or 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`.

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 of -1 means that UOW messages will never expire. **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).

For instructions about configuring unit-of-work parameters on standalone destinations, distributed destinations, or JMS templates using the WebLogic Server Administration Console, see the following sections in the *Oracle WebLogic Server Administration Console Online Help*:

- Configure advanced topic parameters
- Configure advanced queue parameters
- Uniform distributed topics - configure advanced parameters
- Uniform distributed queues - configure advanced parameters
- Configure advanced JMS template parameters

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 UOW consumer code in Example 12-3 shows how a consumer listening on a terminal destination verifies that all component messages sent are contained within the final UOW message.

Example 11-3 Sample Client Code for UOW Terminal Destination

```
{
 msg = qReceiver1.receive();
 if (msg != null)
 {
   count++;
   System.out.println"Message received: " + msg);
   //Check that this one message contains all the messages sent.
   ArrayList msgList = (ArrayList)(((ObjectMessage)msg).getObject());
   numMsgs = msgList.size();
   System.out.println("no. of messages in the msg = " + numMsgs);
 }
} while (msg != null);
```

Message Unit-of-Work Advanced Topics

Learn how Unit-of-Work processes messages in advanced or more complex situations.

- Message Property Handling
- UOW and Uniform Distributed Destinations
- UOW and Store-and-Forward Destinations

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
- User Properties
- User ID

Component Message Heterogeneity

Another method for handling component message heterogeneity is to coerce all component messages into the same value. For example, as mentioned earlier, a mixture of expiration times doesn't make sense. This is the complete list of message properties that are handled in this way:

- Delivery Mode
- Expiration

ReplyTo Message Property

The `ReplyTo` property value is not reflected in the virtual message because it is not used in message selection or sorting and is only useful to the application, therefore it is ignored.

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 back into large giant message on the server before they are pushed to the consumer, rather than streamed.

12

Using Transactions with WebLogic JMS

Learn how to use transactions with WebLogic JMS and learn about JTA user transactions using message driven beans.

> **Note:**
>
> For more information about the JMS classes described in this section, access the latest JMS Specification and Javadoc supplied on the Java Web site at the following location: `http://www.oracle.com/technetwork/java/jms/index.html`.

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 an application commits a transaction, all of the messages it received within the transaction are removed from the messaging system and the messages it sent within the transaction are delivered. If the application rolls back the transaction, then the messages it received within the transaction are returned to the messaging system and messages it sent are discarded.

When a topic subscriber rolls back a received message, the message is redelivered to that subscriber. When a queue receiver rolls back a received message, the message is redelivered to the queue, not the consumer, so that another consumer on that queue can receive the message.

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.

There are three ways to use transactions with JMS:

- If you are using only JMS in your transactions, you can create a *JMS transacted session*.
- If you are mixing other operations, such as EJB, with JMS operations, you should use a *Java Transaction API (JTA) user transaction* in a non-transacted JMS session.

- Use message driven beans.

The following sections explain how to use a JMS transacted session and JTA user transaction.

> **Note:**
>
> When using transactions, it is recommended that you define a session exception listener to handle any problems that occur before a transaction is committed or rolled back, as described in Defining a Connection Exception Listener.
>
> If the `acknowledge()` method is called within a transaction, then it is ignored. If the recover() method is called within a transaction, a `JMSException` is thrown.

Using JMS Transacted Sessions

A JMS transacted session supports transactions that are located within the session.

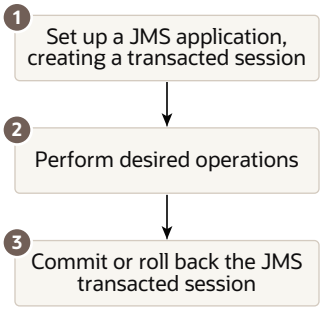
A JMS transacted session's transaction will not have any effect outside of the session. For example, rolling back a session will roll back all sends and receives on that session, but will not roll back any database updates. JTA user transactions are ignored by JMS transacted sessions.

Transactions in JMS transacted sessions are started implicitly, after the first occurrence of a send or receive operation, and chained together; whenever you commit or roll back a transaction, another transaction automatically begins.

Before using a JMS transacted session, the system administrator should adjust the connection factory (Transaction Timeout) and/or session pool (Transaction) attributes, as necessary for the application development environment.

Figure 13-1 shows the steps required to set up and use a JMS transacted session.

Figure 12-1 Setting Up and Using a JMS Transacted Session



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.

If either the `commit()` or `rollback()` methods are issued outside of a JMS transacted session, then a `IllegalStateException` is thrown.

Using JTA User Transactions

The Java Transaction API (JTA) supports transactions across multiple data resources. JTA is implemented as part of WebLogic Server and provides a standard Java interface for implementing transaction management.

You program your JTA user transaction applications using the `javax.transaction.UserTransaction` object, described at `http://www.oracle.com/technetwork/java/javaee/jta/index.html`, to begin, commit, and roll back the transactions. When mixing JMS and EJB within a JTA user transaction, you can also start the transaction from the EJB, as described in Transactions in EJB Applications in *Developing JTA Applications for Oracle WebLogic Server*.

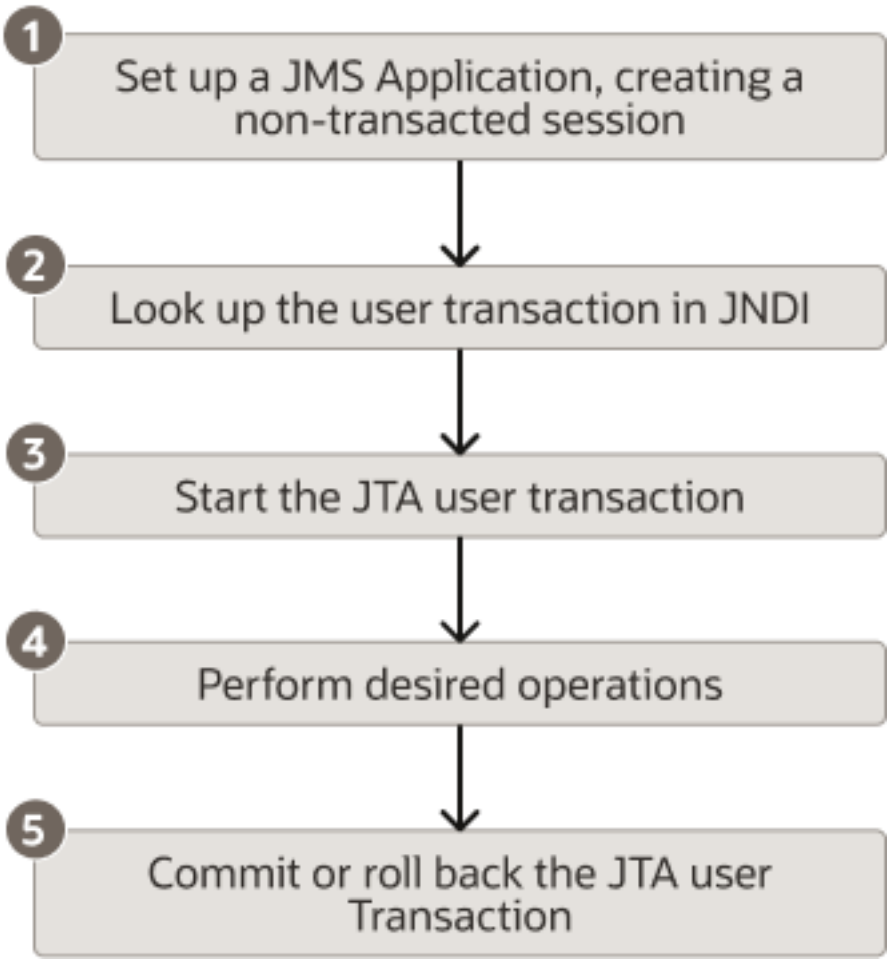
You can start a JTA user transaction after a transacted session has been started; however, the JTA transaction will be ignored by the session and vice versa.

WebLogic Server supports the two-phase commit protocol (2PC), enabling an application to coordinate a single JTA transaction across two or more resource managers. It guarantees data integrity by ensuring that transactional updates are committed in all of the participating resource managers, or are fully rolled back out of all the resource managers, reverting to the state before the start of the transaction.

Before using a JTA transacted session, the system administrator must configure the connection factories to support JTA user transactions by selecting the XA Connection Factory Enabled check box.

Figure 13-2 shows the steps required to set up and use a JTA user transaction.

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
);
```

> **Note:**
>
> When a user transaction is active, the acknowledge mode is ignored.

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.

You can look up the `UserTransaction` object by establishing a JNDI context (`context`) and executing the following code, for example:

```
UserTransaction xact = ctx.lookup("javax.transaction.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 causes WebLogic Server to call the Transaction Manager to complete the transaction, and commit all operations performed during the current transaction. The Transaction Manager is responsible for coordinating with the resource managers to update any databases.

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()`.

JTA User Transactions Using Message Driven Beans

Use message-driven beans to simulate asynchronous message delivery within JTA user transactions.

Because JMS cannot determine which, if any, transaction to use for an asynchronously delivered message, JMS asynchronous message delivery is not supported within JTA user transactions.

However, message— driven beans provide an alternative approach. A message driven bean can automatically begin a user transaction just before message delivery.

See Designing Message-Driven EJBs in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

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.

The following example shows the steps to set up an application by looking up a `javax.transaction.UserTransaction` using JNDI, and beginning and then committing a JTA user transaction. In order for this example to run, the XA Connection Factory Enabled check box must be selected when the system administrator configures the connection factory.

> **Note:**
>
> In addition to this simple JTA User Transaction example, see example provided with WebLogic JTA, located in the *EXAMPLES_HOME*`\wl_server\examples\src\examples\jta\jmsjdbc` directory, where *EXAMPLE_HOME* represents the directory in which the WebLogic Server code examples are configured.

Import the appropriate packages, including the `javax.transaction.UserTransaction` package, at `http://www.oracle.com/technetwork/java/javaee/jta/index.html`.

```
import java.io.*;
import java.util.*;
import javax.transaction.UserTransaction;
import javax.naming.*;
import javax.jms.*;
```

Define the required variables, including the JTA user transaction variable.

```
public final static String JTA_USER_XACT=
       "javax.transaction.UserTransaction";
       .
       .
       .
```

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

You must correctly configure either the Cross— Domain Security or Security Interoperability Mode for all participating domains.

Keep all the domains used by your process symmetric with respect to Cross Domain Security configuration and Security Interoperability Mode. Because both settings are set at the domain level, it is possible for a domain to be in a mixed mode, meaning the domain has both Cross Domain Security and Security Interoperability Mode set. See Configuring Secure Inter-Domain and Intra-Domain Transaction Communication in *Developing JTA Applications for Oracle WebLogic Server*.

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 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.

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*.

In order to share a subscription, durable or non-durable subscriptions must have the Client Id set on their connection factory or connection. Before WebLogic Server 10.3.4.0, a Client ID was exclusively used by one connection at any given time. In this release of WebLogic Server, this restriction is relaxed and a new Client ID Policy is used to restrict or not restrict use of a Client ID. The default policy, `Restricted`, allows only one Client ID to be used by one connection. The `Unrestricted` policy allows multiple connections to use the same client ID. For more information, see How Sharing a Durable Subscription Works.

What is the Subscription Key

A subscription key is used to uniquely identify a subscription. For non-durable subscriptions, the key is composed of the Client ID and Client ID Policy. For durable subscriptions, the key is composed of the Client ID, Client ID Policy, and Subscription Name.

Configuring a Shared Subscription

To configure a shared subscription, you need to configure the Subscription Sharing Policy attribute on the connection factory. Setting the Subscription Sharing Policy to `Sharable` allows subscribers created using a connection factory to 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 a 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. See:

- Configure a connection factory subscription sharing policy in *Oracle WebLogic Server Administration Console Online Help*
- ClientIdPolicy in *MBean Reference for Oracle WebLogic Server*

How Sharing a Non Durable Subscription Works

In order to share a subscription among multiple non durable subscribers, the subscribers have to have a Client ID, which serves to identify the subscription. All subscribers that intend to share a subscription must have the same subscription key (`clientID` and `clientIDPolicy`) on their connection. If Subscription Sharing Policy is set to `SHARABLE`, but the `clientID` is not set on the `Connection`, the subscription is not a shared subscription.

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.
- If a `clientID` is used by an `EXCLUSIVE` subscriber, any current or subsequent subscribers using the same `clientID`, selector, and `noLocal` option is treated as a different subscription.

> **Note:**
>
> It is only possible to have the same `clientID` if the subscriber is created with the same connection instance or a connection using the `UNRESTRICTED` client ID policy).

How a Shared Subscription Policy for a Non durable Subscription Is Determined

The Subscription Sharing Policy for a particular non-durable subscription is dynamically determined by the first active subscriber on the subscription and does not change for the life of the subscription. Any attempt to change the Shared Subscription

Policy for a subscription throws an `InvalidSubscriptionSharingException`, which extends `javax.jms.JMSException`. For example:

- If a non-durable subscription has an `EXCLUSIVE` subscriber on a destination, the subscription is `EXCLUSIVE`, and any attempt to create an additional subscriber using the subscription on the same destination fails with an `InvalidSubscriptionSharingException`, regardless of whether the yet-to-be-created subscriber is `EXCLUSIVE` or `SHARABLE`.
- If a subscription has active subscribers with a `SHARABLE` policy, then the subscription is `SHARABLE`, and any attempt to create a new `EXCLUSIVE` subscriber on the subscription fail with an `InvalidSubscriptionSharingException`.

How a Non durable Subscription Is Closed

After all subscribers that share the same subscription close, the subscription is cleaned up. Specifically, when the *last* subscriber consumer on a shared subscription calls the `close()` method, the subscription and all the associated JMS resources cleaned up.

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.

How Sharing a Durable Subscription Works

In previous releases, the subscription key (`<ClientID, SubscriptionName>`) uniquely identified a subscription within a cluster where the subscription could only exist on a single destination instance or a single member of a DT within the cluster. In this WebLogic Server release, the subscription key becomes `<ClientID, ClientIDPolicy, SubscriptionName>`. All durable subscribers that use the same subscription key share the same subscription if they subscribe to a regular topic, or if they subscribe to the same member of a distributed topic. Multiple subscriptions that use the same subscription key can exist on multiple distributed destination member destinations.

The first subscriber that is created with a particular 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 and they are on the same physical destination.

If a subscription is created with a selector and the `noLocal` option, a subscriber created on the same physical destination using the same subscription key with a different selector and `noLocal` option will:

- Replace the existing durable subscription and clean-up all pending messages that are saved for the durable subscription if there are no active subscribers using this existing subscription.
- Throw an`InvalidSubscriptionSharingException` if there are active subscribers using the same subscription key with a different selector or `noLocal` option.

How a Shared Subscription Policy for a Durable Subscription is Determined

The Subscription Sharing Policy for a particular durable subscription is dynamically determined by the first active subscriber on the subscription and does not change unless all current subscribers close and new subscribers attach with a different policy. Any attempt to

change the policy of a subscription that already has active subscribers throws an `InvalidSubscriptionSharingException`. For example:

- If a durable subscription has an `EXCLUSIVE` subscriber and the Subscription Sharing Policy is `EXCLUSIVE`, any attempt to create an additional subscribers on the subscription throws an `InvalidSubscriptionSharingException`, regardless of whether the yet-to-be-created subscriber is `EXCLUSIVE` or a`SHARABLE`.
- If a durable subscription has active subscribers with a `SHARABLE` policy, the Subscription Sharing Policy is `SHARABLE` and, any attempt to create a new `EXCLUSIVE` subscriber on the subscription throws an `InvalidSubscriptionSharingException`.

> **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.

> **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.

- Applications that use a client ID Policy with a value of `UNRESTRICTED` unsubscribe a durable subscription using the `WLSession.unsubscribe(String name, Topic topic)` extension by supplying the subscription name and the topic or a distributed topic member object.

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`.
- If there are no active consumers on a subscription, then a call to the `unsubscribe()` method deletes the matching durable subscription identified by the subscription key `<ClientID, ClientIDPolicy, SubscriptionName>`.

- 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

When there are subscriptions distributed throughout a cluster, it is possible there are some subscriptions that should were deleted but have not been deleted. These subscriptions are sometimes called "abandoned" subscriptions, and can continue to accumulate messages even though there is no subscriber processing the messages. If the accumulating messages never expire, they can eventually cause the topic to begin throwing resource allocation exceptions (quota exceptions), or if quotas are not configured, then the accumulating message can even cause a server to run out of memory.

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.

To help handle these situations, administrators have the following options to monitor and manage durable subscriptions:

- There is one instance of the `JMSDurableSubscriptionRuntimeMBean` for each durable subscription. Administrators can monitor a topic or UDT using the WebLogic Server Administration Console or by using WLST command line or scripts. See Monitor JMS servers in *Oracle WebLogic Server Administration Console Online Help*.
- To find an abandoned or orphaned durable subscription, the administrator can check the `LastMessagesReceivedTime` on the `JMSDurableSubscriberRuntimeMBean`. The `getLastMessagesReceivedTime()` method returns the last time a message was received by a subscriber from the subscription. Based on this information, together with attributes like the `MessagesPendingCount` or `BytesPendingCount` on the same MBean, the administrator can build a clear picture of the status of a particular durable subscription and take appropriate action, such as cleanup the resources.

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
- One-Copy-Per-Application Design Strategy

One-Copy-Per-Instance Design Strategy

The one-copy-per-instance design strategy is the traditional design pattern and is backward compatible with WebLogic Server releases before 10.3.4.0. One-copy-per-instance has the following characteristics:

- Each instance of an application gets one copy of each message that is published to the topic.
- This pattern is usually best implemented by leveraging an MDB, which sets up policies and subscriptions across a cluster automatically. See Best Practices for Distributed Topics.

One-Copy-Per-Application Design Strategy

The One-Copy-Per-Application design strategy is a design pattern available in WebLogic Server 10.3.4.0 and higher releases. One-copy-per-application design strategyhas the following characteristics:

- This pattern is usually best implemented by leveraging an MDB, which sets up policies and subscriptions across a cluster automatically. See Best Practices for Distributed Topics.
- Each application as a whole (that is all instances of the application together) receives one copy of each message that is published to the DT. That is, each instance only receives a subset of the messages that are sent to the DT
- An `UNRESTRICTED` Client ID Policy
- An `SHARABLE` Subscription Sharing Policy
- Uses the same subscription name if the subscribers are durable
- All consumers subscribe to the same topic instance (or member of a DT)

Considerations When Using JMS 2.0 Shared Subscriptions

JMS 2.0 shared subscriptions internally leverage the proprietary WebLogic shared subscription feature. Therefore, JMS 2.0 and proprietary WebLogic shared subscriptions have similar semantics.

This section provides information about how to use JMS 2.0 shared subscriptions to avoid throwing exceptions:

- When a shared non-durable subscription is created on a distributed topic directly or on a distributed topic member, and if the client ID is not set on the connection, use a connection with an `UNRESTRICTED` client ID Policy.
- When a shared durable subscription is created on a distributed topic directly, either use MDBs or use extensions and subscriptions on members.
- When a shared durable subscription is created on a distributed topic member, and if the client ID is not set on the connection, then use a connection with an `UNRESTRICTED` client ID Policy.

> **Note:**
>
> When the client ID Policy is set to `UNRESTRICTED`, unsubscribe a durable subscription using the `WLSession.unsubscribe(String name, Topic topic)` extension by supplying the subscription name and the topic or a distributed topic member object.

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:

- 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

Before replacing an RDT with a different type of topic, it is important to make sure that pre-existing messages are processed. In addition, any old subscriptions on the RDT should be deleted – or more simply, all store files or database tables should be deleted. If the change needs to occur without restarting a cluster, create a new topic with a different name and delete the old topic.

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:

- If you are using MDBs to consume from a PDT, then each MDB's `topic-message-distribution-mode` attribute will need to be set to `one-copy-per-server` or `one-copy-per-app` if it is not already. The default compatibility `topic-message-distribution-mode` will not work with PDTs - the MDB will generate an exception. 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*.
- Similarly, if you are using the OSA JMS Adapter to consume from a PDT, then it is likely no change is needed.

- If you have `javax.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 Publish/Subscibe 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

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
- Manually Migrating JMS Data to a New Server

Automatic JMS Client Failover

With the automatic JMS client reconnect feature, if a server or network failure occurs, some JMS client objects will transparently failover to use another server instance, as long as one is available.

> **Note:**
>
> The WebLogic JMS automatic reconnect feature is deprecated. The JMS connection factory configuration, `javax.jms.extension.WLConnection` API, and `javax.jms.extension.JMSContext` API for this feature will be removed or ignored in a future release. They do not handle all possible failures and so are not an effective substitute for standard resiliency best practices. Oracle recommends that client applications handle connection exceptions as described in Client Resiliency Best Practices in *Administering JMS Resources for Oracle WebLogic Server*.

With the automatic JMS client reconnect feature, if a fatal server failure occurs, then JMS clients automatically attempt to reconnect to the server when it becomes available.

A network connection failure could be due to transient reasons (a temporary interruption in the network connection) or non-transient reasons (a server bounce or network failure). In such cases, some JMS client objects will try to automatically operate with another server instance in a cluster, or possibly with the host server.

By default, JMS producer session objects automatically try to reconnect to an available server instance without any manual configuration or modifications to the existing client code. If you do not want your JMS producers to be automatically reconnected, then you must explicitly disable this feature either programmatically or administratively.

In addition, JMS consumer session objects can also be configured to automatically attempt to reconnect to an available server, but due to their potentially asynchronous nature, you must explicitly enable this capability using the WebLogic Server Administration Console or public WebLogic JMS APIs.

Related Topics

- Automatic Reconnect Limitations
- Automatic Failover for JMS Producers

- Configuring Automatic Failover for JMS Consumers
- Explicitly Disabling Automatic Failover on JMS Clients
- Best Practices for JMS Clients Using Automatic Failover

Automatic Reconnect Limitations

Automatic reconnect logic can provide a seamless failover for clients in many failure scenarios. However, 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. For instance:

- If the message send operation is idempotent, resend the message.
- Otherwise, your application may need to take some action. For instance, you may need to check if the message is already available on the queue before resending to avoid duplicates.

> **Note:**
>
> If the destination or distributed destination member is unavailable, you will not be able to determine if the message send operation was successful until that member becomes available.

Implicit failover of the following JMS objects is not supported before WebLogic Server 9.2:

- Queue browsers: `javax.jms.QueueBrowser`
- The WebLogic JMS thin client (wljmsclient.jar) does not automatically reconnect.
- Client statistics are reset on each reconnect, which results in the loss historical data for the client.
- Under some circumstances, automatic reconnect is not possible. If it is not possible, an exception is reported.
- Temporary destinations (`javax.jms.TemporaryQueue` and `javax.jms.TemporaryTopic`).

> **Note:**
>
> Temporary destinations may still be accessible after a sever/network failure. This is because temporary destinations are not always on the same server instance as the local connection factory due to server load balancing. Therefore, if a temporary destination survives a server/network failure and a producer continues sending messages to it, an auto-reconnected consumer may or may not be able consume messages from the same temporary destination it was connected to before the failure occurred.

Automatic Failover for JMS Producers

In most cases, JMS producer applications will transparently failover to another server instance if one is available. The following WebLogic JMS producer-oriented objects will attempt to automatically reconnect to an available sever instance without any manual configuration or modification to the existing client code:

- `Connection`
- `Session`
- `MessageProducer`

If you do not want your JMS clients to be automatically reconnected, then you must explicitly disable this feature either programatically or administratively, as described in Explicitly Disabling Automatic Failover on JMS Clients.

Sample Producer Code

In the event of a network failure, the WebLogic JMS client code for message production will try to reconnect to an available server during Steps 3-8 shown in Example 14-1.

Example 14-1 Sample JMS Client Code for Message Production

```
  //set exception listener
1. public void onException(javax.jms.JMSException jsme) {
         connection.setExceptionListener
         // handle the exception, which may require checking for duplicates
         // or sending the message again
         }

2.  Context ctx = create WebLogic JNDI context with credentials etc.
3.   ConnectionFactory cf = ctx.lookup(JNDI name of connection factory)
4.   Destination dest     = ctx.lookup(JNDI name of destination)
     // the following operations recover from network failures
5.   Connection  con      = cf.createConnection()
6.   Session     sess     = con.createSession(no transactions, ack mode)
7.   MessageProducer prod = sess.createProducer(dest)

8.   Loop over:
9.      Message msg = sess.createMessage()
       // try block to handle destination availablitiy scenarios
10.     try {
              prod.send(msg)}
           catch (Some Destination Availability Exception e) {
              //handle the exception, in most cases, the destination or member
              //is not yet available, so the code should try to resend
           }
      //end loop

     // done sending messages
11.   con.close(); ctx.close();
```

The JMS producer will transparently fail-over to another server instance, if one is available. This keeps the client code as simple as listed in Example 15-1and eliminates the need for client code for retrying across network failures.

The WebLogic JMS does not reconnect `MessageConsumer`s by default. For this to automatically occur programmatically, your client application code must call the WebLogic

`WLConnection` extension, with the `setReconnectPolicy` set to `"all"`, as explained in Configuring Automatic Failover for JMS Consumers.

Re usable ConnectionFactory Objects

A `ConnectionFactory` object looked up using JNDI (see Step 1 in Example 14-1 and Example 14-2) is re usable 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 client subsequently connects.

Re usable Destination Objects

A destination object (queue or topic) looked up using JNDI (see Step 2 in Example 14-1 and Example 14-2) is re usable after a server or network failure without requiring another lookup. The same principle applies to producers that send to a distributed destinations, because the client looks up the distributed destination in JNDI, and not the unavailable distributed member.

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

The JMS connection object is used to map one-to-one to a physical network connection between the client JVM and a remote WebLogic Server instance. With the JMS client reconnect feature, the JMS Connection object that the client gets from the `ConnectionFactory.createConnection()` method (see Step 3 in Example 14-1 and Example 14-2) maps in a one-to-one-at-a-time fashion to the physical network connection. One consequence is that while the JMS client continues to use the same Connection object, it could be actually communicating with a different WebLogic Server instance after an implicit failover.

If there is a network disconnection and a subsequent implicit refresh of the connection, then all objects derived from the connection (such as `javax.jms.Session` and `javax.jms.MessageProducer` objects) are also implicitly refreshed. During the refresh, any synchronous operation on the connection or its derived objects that go to the server (such as `producer.send()` or `connection.createSession()`), may block for a period of time before giving up on the connection refresh. This time is configured using the WebLogic Server Administration Console or the `setReconnectBlockingMillis(long)` API in the `weblogic.jms.extension.WLConnection` interface.

The reconnect feature keeps trying to reconnect to the WebLogic Server instance's ConnectionFactory object in the background until the application calls `connection.close()`. The `ReconnectBlockingMillis` parameter is the time-out for a synchronous caller trying to use the connection when the connection in being retried in the background.

If a synchronous call times out without seeing a refreshed connection, then it then behaves in exactly the same way (that is, throws the same Exceptions) as without the implicit reconnect (that is, it will behave as if it was called on a stale connection without the reconnect feature).

The caller can then decide to retry the synchronous call (with a potentially lower quality of service, like duplicate messages), or decide to call `connection.close()` method , which will terminate the background retries for that connection.

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.

  > **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 `javax.jms.Connection.close()`, `javax.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 (see Step 4 in Example 14-1 and Example 14-2). 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 `createMessageProducer()`, after a refresh.

Special Cases for Reconnected Sessions

These sections discuss special cases that can occur when Sessions are reconnected.

- *Transacted Sessions With Pending Commits or Rollbacks* – Operations similar to non-transacted JMS Sessions, transacted JMS sessions are automatically refreshed. However, if there were send or receive operations on a session pending a commit or rollback at the time of the network disconnect, then the first commit call after the Session refresh will fail throwing a `javax.jms.TransactionRolledBackException`. When a JMS session transaction spans a network refresh, the commit for that transaction cannot vouch for the operations done before the refresh as part of that transaction (from an application code perspective).

  After a session refresh, operations like `send()` or `receive()` will not throw an exception; it is only the first commit after a refresh that will throw an exception. However, the first commit after a session refresh will not throw an exception if there were no pending transactional operations in that JMS session at the time of the network disconneciont. In case of `Session.commit()` throwing the exception, the client application code can simply retry all the operations in the transaction again with the same (implicitly refreshed) JMS objects. The stale operations before a refresh will not be committed and will not be duplicated.

- *Pending Unacknowledged Messages* – If a session had unacknowledged messages prior to the session refresh, then the first `WLSession.acknowledge()` call after a refresh throws a `weblogic.jms.common.LostServerException`. This indicates that the `acknowledge()` call may not have removed messages from the server. As a result, the refreshed session may receive duplicate messages that were also delivered before the disconnect.

Reconnected MessageProducer Objects

As described in Reconnected Connection Objects, JMS `MessageProducer` objects are refreshed when their associated JMS connection gets refreshed (see Step 5 in Example 14-1). If producers are non-anonymous, that is, they are specific to a destination object (standalone or distributed destination), then the producer's destination is also implicitly refreshed, as described in Re usable 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.

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

JMS `MessageConsumer` objects that are part of a JMS Connection (through a JMS Session) can be refreshed during a JMS connection refresh (see Step 5 in Example 14-2). However, due to the stateful nature of JMS consumers, as well as their potential asynchronous nature, you must explicitly enable this capability using either the `weblogic.jms.extension.WLConnection` API or the WebLogic Server Administration Console.

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, refreshed consumers does not include `QueueBrowser` clients, which are never refreshed, as described in Automatic Reconnect Limitations.

Sample Consumer Client Code

When Message Consumer refresh is explicitly activated, in the event of a network failure, the WebLogic JMS client code for message consumption will attempt to reconnect during Steps 3-8 in Example 14-2.

Example 14-2 Sample JMS Client Code for Message Consumption

```
0.  Context ctx = create WebLogic JNDI context with credentials etc.
1.  ConnectionFactory cf = ctx.lookup(JNDI name of connection factory)
2.  Destination dest     = ctx.lookup(JNDI name of destination)
     // the following operations recover from network failures
3.  Connection  con      = cf.createConnection()
    (weblogic.jms.extensions.WLConnection)con).setReconnectPolicy("all")
4.  Session     sess     = con.createSession(no transactions, auto ack)
5.  MessageConsumer cons = sess.createConsumer(dest, message selector)
    - also for async consumers : cons.setMessageListener(onMessage impl)
6.  con.start()
7.  Loop over:
      for sync consumers: Message msg = consumer.receive()
      for async consumers (in different thread): onMessage() invoked
8.  con.close(), ctx.close()
```

Note that the connection factory does not refresh `MessageConsumer` objects by default. For this to occur programmatically, your client application code must call the WebLogic `WLConnection` extension, with the `setReconnectPolicy` set to `"all"`, as shown in Step 3 in Example 14-2.

Configuring Automatic Client Refresh Options

The JMS client reconnect API includes the following configuration parameters, which enables you to make some choices that affect the behavior of the reconnect feature for consumers.

Table 14-1 Automatic JMS Client Reconnect Options

Console Label/MBean Attribute	Value	Description
Reconnect Policy `ReconnectPolicy`	• None • Producer (default) • All	Determines which JMS client objects are implicitly refreshed when a network disconnect or server reboot. It only affects the implicit refresh of connections, sessions, producers, and consumers derived from this connection factory. This attribute does not affect Destination or ConnectionFactory objects in the JMS client, since those objects are always refreshed implicitly. Nor does it affect the QueueBrowser object in the JMS client, since that object is never refreshed.
Reconnect Blocking Time `ReconnectBlockingTimeMillis`	6000	Determines how long any synchronous JMS calls, such as `producer.send()`, `consumer.receive()`, and `session.createBrowser()` will block the calling thread before giving up on a JMS client reconnect in progress.
`TotalReconnectPeriodMillis`	-1	Determines how long JMS clients should keep retrying to connect after either the initial network disconnection or the last synchronous JMS call attempt (whichever occurs most recently), before giving up retrying.

For instructions about configuring client parameters on a connection factory using the WebLogic Server Administration Console, see Configure connection factory client parameters in the *Oracle WebLogic Server Administration Console Online Help*. 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

Synchronous consumers use `MessageConsumer.receive()`, `MessageConsumer.receive(timeout)`, and `MessageConsume.receiveNoWait()` methods to consume messages. The first two methods are already expected to potentially block the application code, while the third method is not expected to block the application code. To retain these semantics, the following rules describe interaction of the reconnect feature with the synchronous consumer calls:

- `MessageConsumer.receive()` – If there is a network disconnection during this call, this method can block for up to Reconnect Blocking Time property (described in the configuration section) for a reconnect to go through before throwing an Exception.
- `MessageConsumer.receive(timeout)` – This call will block for the at-most timeout in milliseconds specified by the caller. If the Reconnect Blocking Time property is less than the timeout, then the receive will still block up to the Reconnect Blocking Time setting; if the Reconnect Blocking Time value is more than the timeout, the receive will only block up to timeout.

- `MessageConsumer.receiveNoWait()` – This call will not block if the JMS Connection is in the process of reconnecting. The Reconnect Blocking Time value has no effect on this call.

If these methods eventually reach their respective timeout/wait periods, they all will throw the same Exceptions. as they would reconnect. If a reconnect succeeds while these methods are blocked/called, then these methods will continue returning messages, but with a potentially lowered quality-of-service and with generally similar semantics of receiving messages (like Redelivered messages), as after a recover. The application is notified of this possibility by a Connection ExceptionListener callback with the`LostServerException`. In addition, for non-`AUTO_ACK` acknowledge modes, the first acknowledge call after a refresh will throw a `LostServerException` to notify the application of this possibility.

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 `onMessages` 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

Before to WebLogic Server 9.2, consumers of distributed destinations (DDs) were pinned to a particular destination member of the DD for the life of the pinned consumer. This applies to queue consumers of distributed queues, and non-durable subscribers of distributed topics (durable subscribers are not supported distributed topics).

With MessageConsumer reconnect, DD consumers are also refreshed; however, the refreshed consumer is almost never on the same destination member as the stale consumer. Therefore, even though the application is using the *same* DD consumer across a refresh, it is effectively not pinned to the same destination member across a refresh.

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

Durable subscriptions on standalone topics will not notice any difference due to the client reconnect feature if the topic is still available across a disconnect. The JMS client reconnect framework implicitly refreshes the durable subscriber on that topic and continue from where it was interrupted. Note that if your Reconnect Policy is set to `All`, JMS Connections with a ClientID will also refresh automatically, thus allowing durable subscriptions (which are scoped

by ClientID) to refresh automatically. Connections with a ClientID set will not reconnect for any other Reconnect Policy setting.

> **Note:**
>
> If a JMS Connection has a `ClientID` specified at the time of a network/server failure, then reconnecting that client make take significantly longer than your other clients. For example, in a cluster the JMS server must wait for the WebLogic Server "heartbeat" notification that is broadcast from other members of the cluster, as explained in Failover and Replication in a Cluster in *Administering Clusters for Oracle WebLogic Server*.
>
> WebLogic JMS does not support durable subscriptions on distributed topics, so there is no issue of failover to another distributed topic member during a refresh.

Non Durable Subscriptions and Possible Missed Messages

For consumers that are non-durable subscribers of topics, though the consumption apparently continues successfully across a refresh from an application perspective, it is possible for messages to be published to the topic and dropped (e.g., for lack of consumers) while the reconnect was happening. Missed messages can occur with either synchronous or asynchronous non durable subscribers.

Duplicate Messages

Due to the nature of the consumer refresh feature, there is a possibility of redelivered messages without the client application code calling recover explicitly because a consumer refresh effectively does an implicit equivalent of a recover upon a refresh. This is the main reason why implicit Consumer refresh is not on by default. The semantics of never redelivering a successfully acknowledged message still hold true.

There is also an unlikely case when non-durable subscribers of distributed topics can receive duplicate messages that are not marked redelivered (e.g., when failover happens faster than messages are discarded in topics). This is a consequence of a non-durable subscriber refresh for the distributed topic not being pinned to a topic member across a refresh.

Variations Due to Acknowledge Modes

There will be no difference in the reconnect behaviors of Consumers due to different acknowledge modes. However, the first acknowledge call after a refresh for non-`AUTO_ACK` modes will throw a LostServerException as described earlier to notify user of potential lowered quality of service.

Reconnecting with Migrated JMS Destinations In a Cluster

Consumers will not always reconnect after a JMS server (and its destinations) is migrated to another server in a cluster. If consumers do not get migrated with the destinations, then either an exception is thrown or `onException` will occur to inform the application that the consumer is no longer valid. As a workaround, an application can refresh the consumer either in the exception handler or through `onException`.

Explicitly Disabling Automatic Failover on JMS Clients

If you do not want your JMS clients to be automatically reconnected, then you must explicitly disable this feature either programatically or administratively.

Programmatically

If you do not want your JMS clients to be automatically reconnected, then your applications should call the following code:

```
ConnectionFactory cf = (javax.jms.ConnectionFactory)ctx.lookup
                       (JNDI name of connection factory)
javax.jms.Connection con = cf.createConnection();
((weblogic.jms.extensions.WLConnection)con).setReconnectPolicy("none")
```

For more information about the `setReconnectPolicy` method, see the `weblogic.jms.extension.WLConnection` API.

Administratively

Administrators that do not want JMS clients to automatically reconnect should use the following steps to disable the Reconnect Policy on the JMS connection factory:

1. Follow the directions for getting to the JMS Connection Factory: Configuration: Client pages, see Configure connection factory client parameters in the *Oracle WebLogic Server Administration Console Online Help*.
2. In the Reconnect Policy field, select **None** to disable the JMS client reconnect feature on this connection factory.

   For more information about the Reconnect Policy field, see JMS Connection Factory: Configuration: Client in the *Oracle WebLogic Server Administration Console Online Help*.
3. Click Save.

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,
- Use Transactions to Group Message Work,
- JMS Clients Should Always Call the close() Method,

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 transacted sessions (JMS) or user transactions (JTA) to group related or dependent work, including messaging work, so that either all of the work is completed or none of it is. If a server instance goes down and a message is lost in the middle of a transaction, the entire transaction is rolled back and the application does not need to make a decision for each message after a failure.

> **Note:**
>
> Be aware of transaction commit failures after a server reconnect, which may occur if the transaction subsystem cannot reach all the participants involved in the transaction.

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 the`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.

After a JMS server is properly configured, a JMS server and all of its destinations can migrate to another WebLogic Server within a cluster.

You can manually recover JMS data from a failed WebLogic Server by starting a new server and doing one or more of the tasks in Table 15-3.

> **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	• If the JDBC database store physically exists on the failed server, then migrate the database to a new server and ensure that the JDBC connection pool URL attribute reflects the appropriate location reference. • If the JDBC database does not physically exist on the failed server, access to the database has not been affected , and no changes are required.
Persistent messaging—File Store	If you are using a shared file system, ensure that your file store directories are explicitly configured to reference the shared location (do not depend on the default), otherwise you will need to copy the files to the new server and ensure they have the same directory path as the original server before restarting the migrated file stores. See Using Custom File Stores and File Locations in *Administering the WebLogic Persistent Store*.
Transactions	To facilitate recovery after a failure , WebLogic Server provides the Transaction Recovery Service, which automatically tries to recover transactions when the system startup. The Transaction Recovery Service owns the transaction log for a server. 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:

JMS persistent stores can increase the amount of memory required during initialization of WebLogic Server as the number of stored messages increases. When rebooting WebLogic Server, if initialization fails due to insufficient memory, then increase the heap size of the Java Virtual Machine (JVM) proportionally to the number of messages that are currently stored in the JMS persistent store and try the reboot again.

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*.

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.

By default, all users can access JMS resources in a WebLogic Server or WebLogic cluster. This includes users with remote access, and users running directly in the WebLogic Server or WebLogic cluster itself. To restrict access to WebLogic JMS destinations, you must create security policies on the user's system resources and ensure users have the required roles. See Overview of Securing WebLogic Resources for more information about security roles and policies, and Java Messaging Service (JMS) Resources for policies available to JMS.

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: The security object that represents a user in a WebLogic application.
- Principal and Credentials: A user's user name and password respectively.
- Credentials: Often used to represent the combination of a user's user name and password.

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 Java EE's general security model for EJBs, web applications, and RMI.

It means that WebLogic JMS send and consume operations are:

1. *Checked* using the security subject/role stored implicitly within the current thread.
2. *Not checked* using the user name and password that can be passed to JMS javax.jms `createConnection()` or `createJMSContext()` calls. In other words, the thread's subject from (1) supersedes the user name and password that an application can optionally pass into `createConnection()` or `createJMSContext()`.

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 *Oracle Fusion Middleware Developing Applications with the WebLogic Security Service*. To override thread-based JMS security checking behavior for server-side WebLogic JMS send and consume calls, see Understanding Object Based Security on Server Applications.

Thread-Based Security for Client Applications

For client applications, the current thread's subject is generated and implicitly placed on the thread when the client application creates a JNDI context. A JNDI context can optionally specify credentials by using its `SECURITY_PRINCIPAL` and `Context.SECURITY_CREDENTIALS` properties.

The following code sample puts a subject on the current thread for user *myusername* and password *user_password*:

```
java.util.Hashtable env = new Hashtable();
    env.put(Context.PROVIDER_URL, url);  // typical url: t3://
example.com:7001
    env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
    env.put(Context.SECURITY_PRINCIPAL, "myusername");
    env.put(Context.SECURITY_CREDENTIALS, "user_password");
    javax.naming.InitialContext ic = new InitialContext(env); //
throws an exception if user name/password is incorrect
    // thread now implicitly has subject for the ic user name/password
    ic.close();
    // thread now has original subject from before the ic was created
```

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.

For example:

```
// retrieve the subject that is implicitly store in the current thread
javax.security.auth.Subject subject =
weblogic.security.Security.getCurrentSubject();
```

```
...
// use the given subject to perform an action:
// if the action throws, return an exception
// if the action succeeds, return "OK"
static Object doSomethingAsSubject(javax.security.auth.Subject subject) {
try {
return weblogic.security.Security.runAs(subject,
    new java.security.PrivilegedExceptionAction() {
        public java.lang.Object run() throws Exception {
            //  do something or throw
                 return "OK";
    }});
   } catch (java.security.PrivilegedActionException e) {
    return e;
} catch (Throwable t) {
 return t;
  }
}
```

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 Understanding 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 on Server Applications

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 `createJMSContext()` 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 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 credential is provided when the initial context is created, then, by default, use an anonymous subject for JNDI lookups and WebLogic JMS sends or consumes credentials that are associated with the factory.
      - `ObjectBasedHybrid`: If no credential is provided when the initial context is created, then, by default, use the credential that was on the current thread when the initial context was created for subsequent JNDI lookups and WebLogic JMS sends or consumes credentials that are associated with the factory.
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 `createJMSContext()` 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:

- An OBS initial context only supports `lookup()` calls and will otherwise throw `NotSupported` exceptions. If you need a context that supports other calls, then create a second context that does not enable OBS.

- 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
- Object-Based Security for 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.

> **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:

- Ensures the credentials are dynamically configurable and not hard coded into an application or descriptor file.
- 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. Map the current JNDI location of a JMS connection factory to a local JNDI by:
   - Configuring a foreign JMS Server in a JMS system resource module.
   - Configuring credentials in the foreign JMS Server for users who have the required 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.
>
> After the above requirements are met, WebLogic Server subsequently injects the credentials that you configured in the foreign JMS Server into every outbound send or consume call that originates from the given JMS connection factory.

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.

In addition, in order to secure the internal communication between WebLogic Server instances across domain boundaries, cross-domain security needs to be established between the domains. Using a cross-domain security configuration, WebLogic Server establishes a security role for cross-domain users, and uses the WebLogic Credential Mapping security provider in each domain to store the credentials to be used by the cross-domain users. You can enable cross-domain security in a per domain basis. A cross-domain credential mapping must be configured for each remote domain where internal communications need to be secure. Details about cross-domain security configuration are discussed in Configuring Cross-Domain Security in *Administering Security for Oracle WebLogic Server*. Guidelines are provided for using the cross-domain security in various cross-domain scenarios. See Cross-Domain Security Guidelines.

> **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*.
- If your application or a messaging bridge participates in global transactions that involve more than one WebLogic domain, you need to consider configuring cross-domain security between the two domains. This applies to messaging bridges between two WebLogic domains with an exactly-once QoS. For more details, 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 Java 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 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 Java 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 Java 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.

Here is an example of an application that incorrectly handles security subjects while communicating with two domains. It attempts to receive request messages from a JMS destination in one domain and send response messages to a destination in another domain as indicated in the following code excerpt.

```
// This sample code INCORRECTLY handles security subjects
// in a client that communicates between two domains. It is
// intended to forward JMS messages.

    java.util.Hashtable env = new Hashtable();
    env.put(Context.PROVIDER_URL, domain1_url);  // typical url: t3://
example.com:7001
    env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
    env.put(Context.SECURITY_PRINCIPAL, "mydomain1_username");
    env.put(Context.SECURITY_CREDENTIALS, "mydomain1_password");
    javax.naming.InitialContext ctx = new InitialContext(env);
    // thread now implicitly has subject that is valid in domain1
    final Destination reqDest = (Destination)
ctx.lookup(reqDestJNDI);
    final ConnectionFactory cf = (ConnectionFactory)
ctx.lookup(reqCfJNDI);
    final MessageConsumer consumer =
cf.createContext().createConsumer(reqDest);

    java.util.Hashtable env2 = new Hashtable();
    env2.put(Context.PROVIDER_URL, domain2_url);  // typical url: t3://
example.com:7001
    env2.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
    env2.put(Context.SECURITY_PRINCIPAL, "mydomain2_username");
    env2.put(Context.SECURITY_CREDENTIALS, "mydomain2_password");
    javax.naming.InitialContext ctx2 = new InitialContext(env2);
    // thread now implicitly has subject that is valid in domain2

    final Destination resDest = (Destination)
ctx2.lookup(resDestJNDI);
    final ConnectionFactory cf2 = (ConnectionFactory)
ctx.lookup(resCfJNDI);
    // create JMS producer to send response msg to domain2 now
    MessageProducer producer =
cf2.createConext().createProducer(resDest);

    do {
      // !!The following operation may fail since the current thread
has the subject that is only valid in domain2
      // while the consumer tries to talk to domain1.
      Message msg = consumer.receive(1000);
      if (msg != null) {
        // process msg and generate response message resMsg
        producer.send(resMsg);
```

```
      }
    } while (msg != null);
```

In the above example, the `receive` operation may fail since the current thread has the subject that is only valid in `domain2` while the consumer tries to talk in `domain1`.

To resolve this issue, the JMS client application code needs to use the programming pattern discussed in Thread-Based Security for Client Applications to cache the subjects each time after a new initial context is created, and to restore it on the thread as needed. The following code illustrates the necessary changes to the previous example code.

```
    java.util.Hashtable env = new Hashtable();
    env.put(Context.PROVIDER_URL, domain1_url);  // typical url: t3://
example.com:7001
    env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
    env.put(Context.SECURITY_PRINCIPAL, "mydomain1_username");
    env.put(Context.SECURITY_CREDENTIALS, "mydomain1_password");
    javax.naming.InitialContext ctx = new InitialContext(env);

    // thread now implicitly has subject that is valid in domain1
    // retrieve the subject that is implicitly stored in the current thread
    javax.security.auth.Subject domain1Subject =
weblogic.security.Security.getCurrentSubject();
    final Destination reqDest = (Destination) ctx.lookup(reqDestJNDI);
    final ConnectionFactory cf = (ConnectionFactory) ctx.lookup(reqCfJNDI);

    // create JMS consumer to receive from domain1 now
    MessageConsumer consumer = cf.createContext().createConsumer(reqDest);

    java.util.Hashtable env2 = new Hashtable();
    env2.put(Context.PROVIDER_URL, domain2_url);  // typical url: t3://
example.com:7001
    env2.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
    env2.put(Context.SECURITY_PRINCIPAL, "mydomain2_username");
    env2.put(Context.SECURITY_CREDENTIALS, "mydomain2_password");
    javax.naming.InitialContext ctx2 = new InitialContext(env2);
    // thread now implicitly has subject that is valid in domain2

    final Destination resDest = (Destination)
ctx2.lookup(resDestJNDI);
    final ConnectionFactory cf2 = (ConnectionFactory)
ctx.lookup(resCfJNDI);
     // we can create JMS producer to send response msg to domain2 now
     MessageProducer producer =
cf2.createConext().createProducer(resDest);
     do{
       // use the given subject to perform an action:
       try {
        Message msg = (Message)
weblogic.security.Security.runAs(domain1Subject,
          new.java.security.PrivilegedExceptionAction() {
            public java.lang.Object run()throws Exception {
```

```
                return consumer.receive(1000);

             }});

        } catch (java.security.PrivilegedActionException e) {
           // handle securty exception
        } catch (Throwable t) {
          // handle other throwables
        }
        // the current thread still has the credentials for
domain2
        if (msg ) {
          // process msg and generate response message
resMsg
           producer.send(resMsg);
         }
     }  while (msg != null);
```

Programming Patterns for Using a Foreign JMS Server Between Two WebLogic Domains

> **Note:**
>
> This section does not apply if you are using a foreign JMS server in combination with Java EE resource references, messaging bridges, or MDBs. It is a best practice to use these features when possible because then no special case security handling code is needed as long as cross-domain security is properly configured between the domains.

You can use a foreign JMS server to map JMS resources from one WebLogic domain to another WebLogic domain. When a client or a Java EE application directly looks up the local JNDI name of a foreign JMS destination mapping, a lookup-by-reference is implicitly performed. The remote credentials that are configured in a foreign JMS server are automatically (temporarily) placed on the thread during this lookup by reference, and the thread will resume the local subject that was on the thread before the lookup, after the lookup returns to the client. But note that the remote credentials from a foreign JMS server are not implicitly used for subsequent JMS operations unless you are also using MDBs, messaging bridges, or resource references. As a result, any subsequent JMS calls to a remote domain that do not use these features will cause an access denied error if the JMS resources in the remote domain are protected, even when a foreign JMS configuration contains the correct credentials.

The following code example shows how to use the `JMSDestinationAvailabilityHelper` API to make sure that the correct remote credentials are on the thread when accessing a remote JMS resources using a foreign JMS server without also using a Java EE resource reference, MDB, or messaging bridge. For more information about using the `JMSDestinationAvailabilityHelper`,

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

```
Hashtable h = new Hashtable();
h.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory"
);
h.put(Context.PROVIDER_URL, url);
h.put(Context.SECURITY_PRINCIPAL, user);
h.put(Context.SECURITY_CREDENTIALS, password);
RegistrationHandle handle =
JMSDestinationAvailabilityHelper.getInstance().register(h, destJNDI, new
MyDAHelperListener());
final Context ctx = new InitialContext(h);
final Destination queue = (Destination) ctx.lookup(destJNDI);
final ConnectionFactory cf= (ConnectionFactory) ctx.lookup(cfJNDI);
handle.runAs(
    new PrivilegedExceptionAction(){
     public Object run()throws Exception{
         JMSContext context = cf.createContext();
         for(int i=0; i<msgcount ; i++){
            String msg= text + i;
            context.createProducer().send(queue,msg);
          }
          context.close();
          return null;
         }
        }
  );
  ctx.close();
  handle.unregister();
 }
 private class MyDAHelperListener implements DestinationAvailabilityListener
{
  public void onDestinationsAvailable(String destJNDIName,
List<DestinationDetail> list) {
       // no op for this particular example
  }
  public void onDestinationsUnavailable(String destJNDIName,
List<DestinationDetail> list){
      // no op for this particular example
  }
  public void onFailure(String destJNDIName, Exception exception) {
     // no op for this particular example

  }
}
```

16

WebLogic JMS C API

Understand the requirements, design principles, security considerations and implementation guidelines need to use the WebLogic JMS C API to create C clients that can access WebLogic JMS applications and resources.

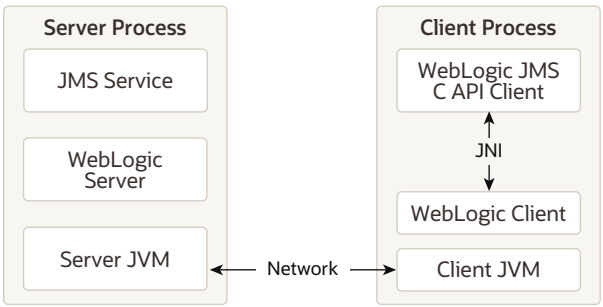
What Is the WebLogic JMS C API?

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 client application then uses the Java Native Interface (JNI), described at `http://docs.oracle.com/javase/8/docs/technotes/guides/jni/index.html`, to access the client-side Java JMS classes. See Figure 16-1.

For this release, the WebLogic JMS C API adheres to the JMS Version 1.1 specification to promote the porting of Java JMS 1.1 code. See the *JMS C API Reference for Oracle WebLogic Server*.

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:
  - 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 handles returned from the WebLogic JMS C API are as thread—safe as their Java counterparts. For example:

- `javax.jms.Session` objects are not thread-safe, and the corresponding WebLogic JMS C API handle, `JmsSession`, is not thread safe.
- `java.jms.Connection` objects are thread-safe, and the corresponding WebLogic JMS C API handle, `JmsConnection`, is thread safe.

As long as concurrency control is managed by the C client application, all objects returned by the WebLogic JMS C API can be used in any thread.

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 `javax.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

When you interoperate between Java code and C code, typically one of the main tasks is converting a C type to a Java type. For example, a `short` type is a two-byte entity in Java as well as in C. The following type conversions that require special handling:

Integer (int)

`Integer (int)` converts to `JMS32I` (4-byte signed value).

Long (long)

`Long (long)` converts to `JMS64I` (8-byte signed value).

Character (char)

`Character (char)` converts to `short` (2-byte Java character).

String

`String` converts to `JmsString`.

Java strings are arrays of 2 -byte characters. In C, strings are generally arrays of 1-byte UTF-8 encoded characters. Pure ASCII strings fit into the UTF-8 specification. For more information about UTF-8 string, see `http://www.unicode.org`. It is inconvenient for C programmers to translate all strings into the 2-byte Java encoding. The `JmsString` structure allows C clients to use native strings or Java strings, depending on the requirements of the application.

`JmsString` supports two kinds of strings:

- Native C `string (CSTRING)`
- `JavaString (UNISTRING)`

A union of the UNISTRING and CSTRING called uniOrC has a character pointer called string that can be used for a NULL terminated UTF-8 encoded C string. The uniOrC union provides a structure called uniString, which contains a void pointer for the string data and an integer length (bytes).

When the stringType element of JmsString is used as input, you should set it to CSTRING or UNISTRING, depending on the type of string input. The corresponding data field contains the string used as input.

The UNISTRING encoding encodes every 2– bytes as a single Java character. The 2-byte sequence is big-endian. Unicode calls this encoding UTF-16BE (as opposed to UTF-16LE, which is a 2-byte sequence that is little-endian). The CSTRING encoding expects a UTF-8 encoded string.

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 JmsMessageGetTextMessage 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

All resources that you allocate must also be disposed of it properly. In Java, garbage collection cleans up all objects that are no longer referenced. However, in C, all objects must be explicitly cleaned up. All WebLogic JMS C API handles given to the user must be explicitly destroyed. Notice that some handles have a verb that ends in Close while others end in Destroy. This convention distinguishes between Java objects that have a close method and those that do not. For example:

- The javax.jms.Session object has a close method so the WebLogic JMS C API has a JmsSessionClose function.
- The javax.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

In Java JMS, closing a connection implicitly closes all subordinate sessions, producers, and consumers. In the WebLogic JMS C API, closing a connection does not close any subordinate sessions, producers, or consumers. After a connection is closed, all subordinate handles are no longer available and need to be explicitly closed.

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

The WebLogic JMS C API supports WebLogic compatibility realm security mode based on a `username` and `password`.

The `username` and `password` must be passed to the initial context in the `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIALS` fields of the hash table used to create the `InitialContext` object.

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.
- 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 client executable dynamically links its JMS C library, include the JMS C API library specific to the platform on which your application will run. JMS C API dynamic libraries can be copied from your WebLogic Server install at:
  - `server/native/aix/ppc/libjmsc.so`
  - `server/native/aix/ppc64/libjmsc.so`
  - `server/native/hpux11/IPF64/libjmsc.so`
  - `server/native/linux/i686/libjmsc.so`
  - `server/native/linux/ia64/libjmsc.so`
  - `server/native/linux/s390x/libjmsc.so`
  - `server/native/linux/x86_64/libjmsc.so`
  - `server/native/solaris/sparc/libjmsc.so`
  - `server/native/solaris/sparc64/libjmsc.so`
  - `server/native/solaris/x64/libjmsc.so`
  - `server/native/solaris/x86/libjmsc.so`
  - `server/native/win/32/jmsc.dll`
  - `server/native/win/64/jmsc.dll`
  - `server/native/win/x64/jmsc.dll`

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:

- Add code in the client to detach the JVM from any C thread that exits and that has previously called into the JMS C-API.

- Do not allow a C thread that has previously called into the JMS C-API to exit before the entire process exits.

The sample Java JNI code shown in Example 16-1 describes how to detach the thread from the JVM.

Example 16-1 Sample Java JNI Code

```
#include <jni.h>

...

JavaVM *jvmList[JVM_LIST_SIZE];
jsize retSize = -1;
jint retVal = JNI_GetCreatedJavaVMs(jvmList, JVM_LIST_SIZE, &retSize);
if ((retVal != 0) || (retSize < 1) ) {
  printf('ERROR: got %d/%d on JNI_getCreatedJavaVMs\n', retVal, retSize);
  return;
}
printf('INFO: got %d/%d on JNI_getCreatedJavaVMs\n', retVal, retSize);
/* The following line assumes that there's exactly one JVM: */
(*(jvmList[0]))->DetachCurrentThread(jvmList[0]);
```

If a program is not directly making JNI calls already, it may be necessary to add compiler and linker parameters for access to the Java JNI libraries. For example, in MicroSoft Visual C++, do the following:

- Add `-I$(JAVA_HOME)/include` and `-I$(JAVA_HOME)/include/win32` to the compile
- Add `$(JAVA_HOME)/lib/jvm.lib` to the link

A

Server Session Pools (Deprecated)

Learn how to configure and use Server Session Pools, a deprecated JMS facility for defining a server-managed pool of server sessions. This facility enables an application to process messages concurrently with a deprecated release of WebLogic Server.

Defining Server Session Pools

> **Note:**
>
> Session pools are used rarely, because they are not a required part of the Java 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 Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

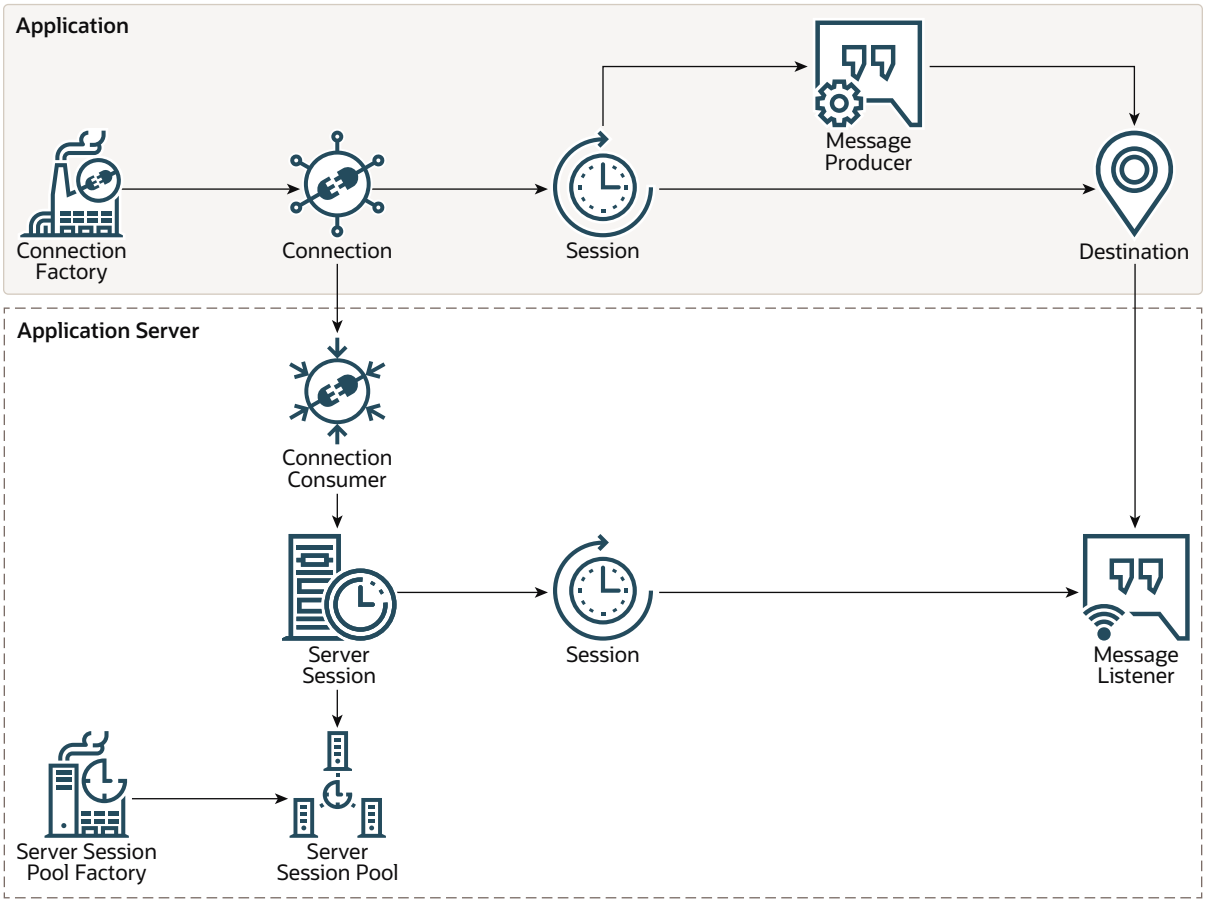
WebLogic JMS implements an optional JMS facility for defining a server-managed pool of server sessions. This facility enables an application to process messages concurrently.

The server session pool:

- Receives messages from a destination and passes them to a server-side message listener that you provide to process messages. The message listener class provides an `onMessage()` method that processes a message.
- Processes messages in parallel by managing a pool of JMS sessions, each of which executes a single-threaded `onMessage()` method.

Figure A-1shows the server session pool facility, and the relationship between the application and the application server components.

Figure A-1 Server Session Pool Facility

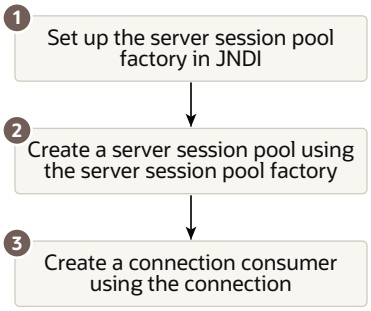


As shown in the Figure A-1, the application provides a single-threaded message listener. The connection consumer, implemented by JMS on the application server, performs the following tasks to process one or more messages:

1. Gets a server session from the server session pool.
2. Gets the server session's session.
3. Loads the session with one or more messages.
4. Starts the server session to consume messages.
5. Releases the server session back to the pool when it has finished processing messages.

Figure A-2shows the steps required to prepare for concurrent message processing.

Figure A-2 Preparing for Concurrent Message Processing



Applications can use other application server providers' session pool implementations within this flow. Server session pools can also be implemented using message-driven beans. For information about using message driven beans to implement server session pools, see Message-Driven EJBs in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

If the session pool and connection consumer were defined during configuration, then you can skip this section. For more information about configuring server session pools and connection consumers, see Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*.

Currently, WebLogic JMS does *not* support the optional `TopicConnection.createDurableConnectionConsumer()` operation. For more information about this advanced JMS operation, refer to the JMS Specification, described at `http://www.oracle.com/technetwork/java/jms/index.html`.

Step 1: Look Up the Server Session Pool Factory in JNDI

You use a server session pool factory to create a server session pool.

WebLogic JMS defines one `ServerSessionPoolFactory` object, by default: `weblogic.jms.extensions.ServerSessionPoolFactory:<name>`, where <*name*> specifies the name of the JMS server to which the session pool is created.

After it is configured, you can look up a server session pool factory by first establishing a JNDI context (`context`) using the `NamingManager.InitialContext()` method, at `http://docs.oracle.com/javase/6/docs/api/javax/naming/InitialContext.html#InitialContext()`. For any application other than a servlet application, you must pass an environment used to create the initial context. For more information, see the `NamingManager.InitialContext()` Javadoc, at `http://docs.oracle.com/javase/6/docs/api/javax/naming/InitialContext.html#InitialContext()`.

After the context is defined, to look up a server session pool factory in JNDI, use the following code:

```
factory = (ServerSessionPoolFactory) context.lookup(<ssp_name>);
```

The `<ssp_name>` specifies a qualified or non-qualified server session pool factory name.

See ServerSessionPoolFactory or the `weblogic.jms.extensions.ServerSessionPoolFactory` Javadoc.

Step 2: Create a Server Session Pool Using the Server Session Pool Factory

You can create a server session pool for use by queue (Point-toPoint) or topic (Publish/Subscribe) connection consumers, using the `ServerSessionPoolFactory` methods described in the following sections.

For more information about server session pools, see ServerSessionPool or the `javax.jms.ServerSessionPool` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ServerSessionPool.html`.

Create a Server Session Pool for Queue Connection Consumers

The `ServerSessionPoolFactory` provides the following method for creating a server session pool for queue connection consumers:

```
public ServerSessionPool getServerSessionPool(
 QueueConnection connection,
 int maxSessions,
 boolean transacted,
 int ackMode,
 String listenerClassName
) throws JMSException
```

You must specify the queue connection associated with the server session pool, the maximum number of concurrent sessions that can be retrieved by the connection consumer (to be created in step 3), whether or not the sessions are transacted, the acknowledge mode (applicable for non-transacted sessions only), and the message listener class that is instantiated and used to receive and process messages concurrently.

For more information about the `ServerSessionPoolFactory` class methods, see the `weblogic.jms.extensions.ServerSessionPoolFactory` Javadoc. For more information about the `ConnectionConsumer` class, see the `javax.jms.ConnectionConsumer` Javadoc, described at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionConsumer.html`.

Create a Server Session Pool for Topic Connection Consumers

The `ServerSessionPoolFactory` provides the following method for creating a server session pool for topic connection consumers:

```
public ServerSessionPool getServerSessionPool(
 TopicConnection connection,
 int maxSessions,
 boolean transacted,
 int ackMode,
 String listenerClassName
) throws JMSException
```

You must specify the topic connection associated with the server session pool, the maximum number of concurrent sessions that can be retrieved by the connection (to be created in step 3), whether or not the sessions are transacted, the acknowledge mode (applicable for non-transacted sessions only), and the message listener class that is instantiated and used to receive and process messages concurrently.

For more information about the `ServerSessionPoolFactory` class methods, see the `weblogic.jms.extensions.ServerSessionPoolFactory` Javadoc. For more information about the `ConnectionConsumer` class, see the `javax.jms.ConnectionConsumer` Javadoc, described at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionConsumer.html`.

Step 3: Create a Connection Consumer

You can create a connection consumer for retrieving server sessions and processing messages concurrently using one of the following methods:

- Configuring the server session pool and connection consumer during the configuration, as described in Configuring Basic JMS System Resources in *Administering JMS Resources for Oracle WebLogic Server*.
- Including in your application the Connection methods described in the following sections.

For more information about the `ConnectionConsumer` class, see ConnectionConsumer or the `javax.jms.ConnectionConsumer` Javadoc, described at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionConsumer.html`.

Create a Connection Consumer for Queues

The `QueueConnection` provides the following method for creating connection consumers for queues:

```
public ConnectionConsumer createConnectionConsumer(
 Queue queue,
 String messageSelector,
 ServerSessionPool sessionPool,
 int maxMessages
) throws JMSException
```

You must specify the name of the associated queue, the message selector for filtering messages, the associated server session pool for accessing server sessions, and the maximum number of messages that can be assigned to the server session simultaneously. For information about message selectors, see Filtering Messages.

For more information about the `QueueConnection` class methods, see the `javax.jms.QueueConnection` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/QueueConnection.html`. For more information about the `ConnectionConsumer` class, see the `javax.jms.ConnectionConsumer` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionConsumer.html`.

Create a Connection Consumer for Topics

The `TopicConnection` provides the following two methods for creating `ConnectionConsumers` for topics:

```
public ConnectionConsumer createConnectionConsumer(
 Topic topic,
 String messageSelector,
 ServerSessionPool sessionPool,
 int maxMessages
) throws JMSException

public ConnectionConsumer createDurableConnectionConsumer(
```

```
 Topic topic,
 String messageSelector,
 ServerSessionPool sessionPool,
 int maxMessages
) throws JMSException
```

For each method, you must specify the name of the associated topic, the message selector for filtering messages, the associated server session pool for accessing server sessions, and the maximum number of messages that can be assigned to the server session simultaneously. For information about message selectors, see Filtering Messages.

Each method creates a connection consumer; but, the second method also creates a durable connection consumer for use with durable subscribers. For more information about durable subscribers, see Setting Up Durable Subscriptions.

For more information about the `TopicConnection` class methods, see the `javax.jms.TopicConnection` Javadoc, at https://javaee.github.io/javaee-spec/javadocs/javax/jms/TopicConnection.html. For more information about the `ConnectionConsumer` class, see the `javax.jms.ConnectionConsumer` Javadoc, at `https://javaee.github.io/javaee-spec/javadocs/javax/jms/ConnectionConsumer.html`.

Example: Setting Up a PTP Client Server Session Pool

The following example shows how to set up a server session pool for a JMS client. The `startup()` method is similar to the `init()` method in the `examples.jms.queue.QueueSend` example, as described in Example: Setting Up a Point-to-Point JMS Application Using the Classic API. This method also sets up the server session pool.

The following illustrates the `startup()` method, with comments highlighting each setup step.

Include the following package on the import list to implement a server session pool application:

Define the session pool factory static variable required for the creation of the session pool.

```
private final static String SESSION_POOL_FACTORY=
       "weblogic.jms.extensions.ServerSessionPoolFactory:examplesJMSServer";

private QueueConnectionFactory qconFactory;
private QueueConnection qcon;
private QueueSession qsession;
private QueueSender qsender;
private Queue queue;
private ServerSessionPoolFactory sessionPoolFactory;
private ServerSessionPool sessionPool;
private ConnectionConsumer consumer;
```

Create the required JMS objects.

```
public String startup(
 String name,
 Hashtable args
) throws Exception
```

```
{
 String connectionFactory = (String)args.get("connectionFactory");
 String queueName = (String)args.get("queue");
 if (connectionFactory == null || queueName == null) {
        throw new  IllegalArgumentException("connectionFactory="+connectionFactory+
                     ", queueName="+queueName);
  }
 Context ctx = new InitialContext();
 qconFactory = (QueueConnectionFactory)
       ctx.lookup(connectionFactory);
 qcon =qconFactory.createQueueConnection();
 qsession = qcon.createQueueSession(false,
       Session.AUTO_ACKNOWLEDGE);
 queue = (Queue) ctx.lookup(queueName);
 qcon.start();
```

Step 1 Look Up the Server Session Pool Factory

Look up the server session pool factory in JNDI.

```
sessionPoolFactory = (ServerSessionPoolFactory)
                ctx.lookup(SESSION_POOL_FACTORY);
```

Step 2 Create a Server Session Pool

Create a server session pool using the server session pool factory, as follows:

```
sessionPool = sessionPoolFactory.getServerSessionPool(qcon, 5,
        false, Session.AUTO_ACKNOWLEDGE,
        examples.jms.startup.MsgListener);
```

The code defines the following:

- `qcon` is the queue connection associated with the server session pool
- `5`is the maximum number of concurrent sessions that can be retrieved by the connection consumer (to be created in step 3)
- Sessions will be non-transacted (`false`)
- `AUTO_ACKNOWLEDGE` is the acknowledge mode
- The `examples.jms.startup.MsgListener` will be used as the message listener that is instantiated and used to receive and process messages concurrently.

Step 3 Create a Connection Consumer

Create a connection consumer, as follows:

The code defines the following:

```
consumer = qcon.createConnectionConsumer(queue, "TRUE",
       sessionPool, 10);
```

- `queue` is the associated queue
- `TRUE` is the message selector for filtering messages
- `sessionPool` is the associated server session pool for accessing server sessions

- *10* is the maximum number of messages that can be assigned to the server session simultaneously

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

Example: Setting Up a Publish/Subscribe Client Server Session Pool

The following example shows how to set up a server session pool for a JMS client. The `startup()` method is similar to the `init()` method in the `examples.jms.topic.TopicSend` example, as described in Example: Setting Up a Publish-Subscribe JMS Application Using the Classic API. It also sets up the server session pool.

The following shows the `startup()` method, with comments highlighting each setup step.

Include the following package on the import list to implement a server session pool application:

```
import weblogic.jms.extensions.ServerSessionPoolFactory
```

Define the session pool factory static variable required for the creation of the session pool.

```
private final static String SESSION_POOL_FACTORY=
       "weblogic.jms.extensions.ServerSessionPoolFactory:examplesJMSServer";

private TopicConnectionFactory tconFactory;
private TopicConnection tcon;
private TopicSession tsession;
private TopicSender tsender;
private Topic topic;
private ServerSessionPoolFactory sessionPoolFactory;
private ServerSessionPool sessionPool;
private ConnectionConsumer consumer;
```

Create the required JMS objects.

```
public String startup(
 String name,
 Hashtable args
) throws Exception

{
 String connectionFactory = (String)args.get("connectionFactory");
 String topicName = (String)args.get("topic");
 if (connectionFactory == null || topicName == null) {
        throw new
IllegalArgumentException("connectionFactory="+connectionFactory+
                     ", topicName="+topicName);
  }
 Context ctx = new InitialContext();
 tconFactory = (TopicConnectionFactory)
       ctx.lookup(connectionFactory);
 tcon = tconFactory.createTopicConnection();
 tsession = tcon.createTopicSession(false,
       Session.AUTO_ACKNOWLEDGE);
```

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

Step 1

Look up the server session pool factory in JNDI.

```
sessionPoolFactory = (ServerSessionPoolFactory)
                ctx.lookup(SESSION_POOL_FACTORY);
```

Step 2 Create a Server Session Pool

Create a server session pool using the server session pool factory, as follows:

```
sessionPool = sessionPoolFactory.getServerSessionPool(tcon, 5,
        false, Session.AUTO_ACKNOWLEDGE,
        examples.jms.startup.MsgListener);
```

The code defines the following:

- `tcon` as the topic connection associated with the server session pool
- `5` as the maximum number of concurrent sessions that can be retrieved by the connection consumer (to be created in step 3)
- Sessions will be non-transacted (`false`)
- `AUTO_ACKNOWLEDGE` as the acknowledge mode
- The `examples.jms.startup.MsgListener` will be used as the message listener that is instantiated and used to receive and process messages concurrently.

Step 3

Create a connection consumer, as follows:

```
consumer = tcon.createConnectionConsumer(topic, "TRUE",
      sessionPool, 10);
```

The code defines the following:

- `topic` as the associated topic
- `TRUE` as the message selector for filtering messages
- `sessionPool` as the associated server session pool for accessing server sessions
- `10` as the maximum number of messages that can be assigned to the server session simultaneously

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

B

FAQs: Integrating Remote JMS Providers

The Java EE standards for JMS (messaging), JTA (transaction), and JNDI (naming) work together to provide reliable Java-to-Java messaging between different host machines and even different vendors. Oracle WebLogic Server provides a variety of tools that leverage these APIs to help integrate remote JMS providers into a local application.

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 stand alone WebLogic server or outside WebLogic server cluster. The remote JMS server can be a WebLogic or a non-WebLogic (foreign) JMS server.

Q. What is JNDI?

A. Java Naming and Directory Interface (JNDI) is a Java EE lookup service that maps names to services and resources. JNDI provides a directory of advertised resources that exist on a particular stand alone (non-clustered) WebLogic server or within a WebLogic server cluster. Examples of these resources include JMS connection factories, JMS destinations, JDBC (database) data sources, and application EJBs.

A client connecting to WebLogic Server in a WebLogic cluster can transparently reference any JNDI advertised service or resource hosted on any WebLogic Server within the cluster. The client doesn't require explicit knowledge of which particular WebLogic Server in the cluster hosts a desired resource.

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 connection-id?

A. JMS connection-IDs are used to name JMS client connections. Durable subscribers require named connections, otherwise connections are typically unnamed. Note that within a clustered set of servers or stand alone server, only one JMS client connection may use a particular named connection at a time. An attempt to create new connection with the same name as an existing connection will fail.

Q. What is the difference between a JMS topic and a JMS queue?

A. JMS queues deliver a message to one consumer, while JMS topics deliver a copy of each message to each consumer.

Q. What is a topic subscription?

A. A topic subscription can be thought of as an internal queue of messages waiting to be delivered to a particular subscriber. This internal queue accumulates copies of each message published to the topic after the subscription was created. Conversely, it does not accumulate messages that were sent before the subscription was created. Subscriptions are not sharable, only one subscriber may subscribe to a particular subscription at a time.

Q. What is a non-durable topic subscriber?

A. A non durable subscriber creates unnamed subscriptions that exist only for the life of the JMS client. Messages in a non durable subscription are never persisted—even when the message's publisher specifies a persistent quality of service (QOS). Shutting down a JMS server terminates all non durable subscriptions.

Q. What is a durable subscriber?

A. A durable subscriber creates named subscriptions that continue to exist even after the durable subscriber exits or the server reboots. A durable subscriber connects to its subscription by specifying the topic-name, connection-ID, and subscriber-ID. Together, the connection-id and subscriber-id uniquely name the subscriber's subscription within a cluster. A copy of each persistent message published to a topic is persisted to each of the topic's durable subscriptions. In the event of a server failure and restart, durable subscriptions and their unconsumed persistent messages are recovered.

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. In Java EE, the terms JTA transaction, XA transaction, user transaction, and global transaction are often used interchangeably to refer to a single global transaction. This type of transaction can include operations on multiple different *XA capable* resources and different resource types. A JTA transaction is always associated with the current thread, and can be passed from server to server as one application calls another. A common example of an XA transaction is one that includes both a WebLogic JMS operation and a JDBC (database) operation.

Q. What is a local transaction?

A. A JMS local transaction is a transaction in which only a single resource or service can participate. A JMS local transaction is associated with a particular JMS session where the destinations of a single vendor participate. Unlike XA transactions, a database operation can not participate in a JMS local transaction.

Q. How does JMS provide local transactions?

A. Local transactions are enabled by a JMS specific API called `transacted sessions`. For vendors other than WebLogic JMS, the scope of a transacted session is typically limited to a single JMS server. In WebLogic JMS, multiple JMS operations on multiple destinations within an entire cluster can participate in a single transacted session's transaction. In other words, it is scoped to a WebLogic cluster and no remote JMS provider to the JMS session's cluster can participate in a transaction.

Q. Are JMS local transactions useful for integration purposes?

A. Local transactions are generally not useful for integration purposes because they are limited in scope to a single resource, typically a messaging or database server.

Q. What is Automatic Transaction Enlistment?

A. Operations on resources such as database servers or messaging servers participate in a Java EE JTA transaction provided that:

- The resource is XA transaction capable
- The resource was enlisted with the current transaction
- The client library used to access the resource is transaction aware (XA enabled).

Automatic participation of operations on an XA capable resource in a transaction is technically referred to as automatic enlistment.

- WebLogic clients using XA enabled WebLogic APIs automatically enlist operation in the current thread's JTA transaction. Examples of XA enabled WebLogic clients include WebLogic JMS XA enabled (or user transaction enabled) connection factories, and JDBC connection pool data sources that are global transaction enabled.
- Foreign (non-WebLogic) JMS clients do not automatically enlist in the current JTA transaction. These clients must either go through an extra step of programmatically enlisting in the current transaction, or use WebLogic provided features that wrap the foreign JMS client and automatically enlist when the foreign JMS client is accessed via wrapper APIs.

JMS features that provide automatic enlistment for foreign vendors are:

- Message-Driven EJBs
- JMS resource-reference pools
- Messaging Bridges

To determine if a non-WebLogic vendor's JMS connection factory is XA capable, check the vendor documentation. Remember, support for transacted sessions (local transactions) does not imply support for global/XA transactions.

How to Integrate with a Remote Provider

Q. What does a JMS client do to communicate with a remote JMS provider?

A. To communicate with any JMS provider, a JMS client must perform the following steps:

1. Look up a JMS connection factory object and a JMS destination object using JNDI
2. Create a JMS connection using the connection factory object
3. Create message consumers or producers using the JMS connection and JMS destination objects.

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 topic subscribers: The connection-id and subscriber-id names that uniquely identify them. Message Driven EJBs provide default values for these values based on 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.
- 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.

  If the JMS application requires transactions, the connection factory must be XA capable. WebLogic documentation refers to XA capable factories as user transactions enabled.

  By default, WebLogic servers automatically provide three non-configurable connection factories:
  - `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.

Additional WebLogic JMS connection factories must be explicitly configured.

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 Java 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. Remote and local JMS resources, such as client connections and sessions, are often pooled to improve performance. Message— driven EJBs automatically pool their internal JMS consumers. JMS consumers and producers accessed through resource-references are also automatically pooled.

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 use of the remote provider's JMS client	Yes for a WebLogic server provider. Other providers must perform enlistment programmatically.	No. Can be done programmatically.
Messaging Bridge	Yes	N/A
Foreign JMS Server Definition	No. To get automatic enlistment, use in conjunction with a JMS resource reference or MDB.	No. To get resource pooling, use in conjunction with a JMS resource reference or MDB.
JMS Resource Reference	Yes	Yes
Message Driven EJBs	Yes	Yes
SAF Client	N/A	N/A
SAF	Yes	N/A

Best Practices When Integrating with Remote Providers

Q. How do I receive messages from a remote a 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 Messaging 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 the destination is not provided by WebLogic Server, and you to include operations on the destination in a global transaction, use a server proxy to encapsulate JMS operations on the foreign vendor in an EJB. Applications running on WebLogic Server have facilities to enlist non-WebLogic JMS providers that are transaction (XA) capable with the current transaction.

If you need store-and-forward capability, consider sending to local destinations and using messaging bridges to forward the message to the foreign destination. See:

- Using a Messaging Bridge
- Using WebLogic Store-and-Forward
- Using WebLogic JMS SAF Client

Another option is to simply use the remote vendor's JNDI and JMS API directly or configuring foreign JMS providers to avoid hard-coding references to them. You must add the foreign provider's class libraries to the client's class-path.

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 WebLogic Server Administration Console, 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. Resource references are specified by servlet and EJB application developers and packaged with an application. They are easy-to-use and provide a level of indirection that lets applications reference JNDI names defined in an EJB descriptor rather than hard coding JNDI names directly into application source code.

JMS resource-references provide two additional features:

- Automatic pooling of JMS resources when those resources are closed by the application.
- Automatic enlistment of JMS resources with the current transaction, even for non-WebLogic JMS providers.

Inside an EJB or servlet application code, use a JMS resource references by including resource-ref elements in the deployment descriptors and then use a JNDI context to look them up using the syntax `java:comp/env/jms/<reference name>`.

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.

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. For non-transactional cases, do not use a global transaction (XA) capable connection factory. This will affect messaging performance. If you do, the resource reference will

automatically begin and commit an internal transaction for each messaging operation. 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. The WebLogic Store-and-Forward (SAF) Service should be used when forwarding JMS messages between WebLogic Server 9.0 or later domains. The SAF service can deliver messages:

- Between two stand-alone server instances
- Between server instances in a cluster

  Across two clusters in a domain
- Across separate domains

Q. When can't I use WebLogic Store-and-Forward?

A. You can't use the WebLogic Store-and-Forward service 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
- Forwarding messages to prior releases of WebLogic Server. See Using a Messaging Bridge
- Interoperating with third-party JMS products (for example, MQSeries) See Using a Messaging Bridge.
- When using temporary destinations with the `JMSReplyTo` field to return a response to a request
- Environment with low tolerance for message latency. SAF increases latency and may lower throughput

Using WebLogic JMS SAF Client

Q. What is the WebLogic JMS SAF Client?

A. The JMS SAF Client feature extends the JMS store-and-forward service introduced in WebLogic Server 9.0 to standalone JMS clients. Now JMS clients can reliably send messages to server-side JMS destinations, even when the client cannot reach a

destination (for example, due to a temporary network connection failure). While disconnected from the server, messages sent by a JMS SAF client are stored locally on the client file system and are forwarded to server-side JMS destinations when the client reconnects. See Reliably Sending Messages Using the JMS SAF Client.

Q. When should I use the WebLogic JMS SAF Client?

A. Use when forwarding JMS messages to WebLogic Server 9.0 or later domains.

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

Messaging bridges can be configured to use transactions to ensure exactly-once message forwarding from any XA capable (global transaction capable) JMS provider to another.

Q. When should I use a messaging bridge?

A. Typically, messaging bridges are used to provide store-and-forward high availability design requirements. A messaging bridge is configured to consume from a sender's local destination and forward it to the sender's actual target remote destination. This provides high availability because the sender is still able to send messages to its local destination even when the target remote destination is unreachable. When a remote destination is not reachable, the local destination automatically begins to store messages until the bridge is able to forward them to the target destination when the target becomes available again.

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.
- Forward messages between WebLogic 9.0 domains: Use WebLogic Store-and-Forward. 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 effect any 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 Messaging Beans

Q. What is a Message Driven EJB (MDB)?

A. Message Driven EJBs are EJB containers that internally use standard JMS APIs to asynchronously receive messages from local, remote, or foreign JMS destinations and then call application code to process the messages. MDBs have the following characteristics:

- Automatically connects to a source destination and automatically retries connecting if the remote destination is inaccessible.
- Support automatic enlistment of the received messages in container managed transactions, even when the JMS provider is not WebLogic.
- Automatically pool their internal JMS connections, sessions, and consumers.
- A MDB's source destination, URL, and connection factory are configured in the EJB and WebLogic descriptors which are packaged as part of an application.
- 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 Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

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.
- Use care when configuring a MDB for container managed transactions. A MDB supports container managed XA transactions when a MDB's descriptor files have `transaction-type` of `Container` and a `trans-attribute` of `Required` and the JMS connection factory is XA enabled. Failure to follow these steps will result in the MDB being non-transactional. The default WebLogic Server setting for a MDB connection factory is XA enabled. The MDB automatically begins a transaction and automatically enlists the received message in the transaction.

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 stand alone clients. AQ JMS uses a database connection and stored JMS messages in a database accessible to an entire WebLogic Server cluster, enabling the use of database features and tooling for data manipulating 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*.

C

How to Look Up a Destination

Learn how to use JNDI and a Create Destination Identifier to look up a message destination.

- Use a JNDI Name
- Use a Create Destination Identifier
- Examples of Syntax Used to Look Up Destinations

> **Note:**
>
> For information about how to configure JMS resources, see Understanding JMS Resource Configurationin *Administering JMS Resources for Oracle WebLogic Server*

Use a JNDI Name

The recommended way to lookup any type of destination is to use JNDI. You can look up a destination by establishing a JNDI context (context) and executing one of the following commands, for PTP or Pub/Sub messaging, respectively:

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

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

The *Dest_name* argument specifies the destination's JNDI name defined during configuration. See Using a JNDI Name and Examples of Syntax Used to Look Up Destinations.

Use a Create Destination Identifier

Create Destination Identifier (CDI) is a less common method to lookup a destination or member of a distributed destination that does not use JNDI. CDI uses one of the following `QueueSession` or `TopicSession` methods to reference a queue or topic, respectively:

```
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

> **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.

Default WebLogic CDI Syntax

Default WebLogic CDI Syntax is a string which contains a JMS server name, module, and the destination configuration name. See Examples of Syntax Used to Look Up Destinations.

Custom WebLogic CDI Syntax

In addition to the default CDI syntax, WebLogic JMS provides the JMSCreateDestinationIdentifier as an additional configuration parameter of a Destination or Uniform Distributed Destination. This enables you to configure a unique reference name when there is more than one queue or topic defined (in one or more modules) with the same value for the default CDI syntax. In other words, it is useful for differentiating two different destinations in two different modules that have the same default CDI name. See Examples of Syntax Used to Look Up Destinations

This name must be unique within the scope of the JMS server to which this destination is targeted. However, it does not need to be unique within the scope of the entire JMS module. For example, two queues can have the same CDI name as long as those queues are targeted to different JMS servers.

> **Note:**
>
> Because, this name must be unique within the scope of a JMS server, verify whether other JMS modules may contain destination names that conflict with this name. It is the responsibility of the deployer to resolve the destination names targeted to JMS servers.

Server Affinity When Looking Up Destinations

The `createTopic()` and `createQueue()` methods also allow a "`./Destination_Name`" syntax to indicate server affinity when looking up destinations. This will locate destinations that are locally deployed in the same JVM as the JMS connection's connection factory host. If the name is not on the local JVM an exception is thrown, even though the same name might be deployed on a different JVM.

An application might use this convention to avoid hard-coding the server name when using the `createTopic()` and `createQueue()` methods so that the code can be reused on different JMS servers without requiring any changes.

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:

- Non distributed Destinations
- Uniform Distributed Destinations
- Weighted Distributed Destinations

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
- CDI Syntax for Non distributed destinations

JNDI Syntax for Non distributed Destinations

Most applications use JNDI instead of CDI to lookup 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 provides examples of the syntax used to reference a non-distributed destination using the`createQueue` or `createTopic`method using CDI:

- When using the default CDI, a string defined by:

  ```
  JMS_Server_Name/JMS_Module_Name!Destination_Name
  ```

- When using the default CDI in an interop module, a string defined by:

  ```
  JMS_Server_Name/interop-jms!Destination_Name
  ```

- When a custom CDI is configured, a string defined by:

  ```
  JMS_Server_Name/CDI_Name
  ```

> **Note:**
>
> When using server affinity (replacing `JMS_Server_Name` with "."), the search is restricted to the JMS connection host rather than the entire cluster.
>
> To reference destination in releases earlier than WebLogic 9.0 Server , use a string defined by *`JMS_Server_Name!Destination_Name`* (for example, `myjmsserver!mydestination`).

Uniform Distributed Destinations

The following section provides examples of the syntax used to reference Uniform Distributed Destinations (UDDs):

- JNDI Syntax for UDDs
- CDI Syntax for UDDs

JNDI Syntax for UDDs

Most applications use JNDI instead of CDI to lookup destinations. The following section provides examples how to reference an individual member or logical UDD using JNDI

- For a logical UDD, a string defined by:

  ```
  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
  ```

  Where the `wl-server-name` in this case is the configured name of a WebLogic Server in a configured cluster, or is the dynamic-server server-name-prefix appended with a server number in a dynamic cluster.

CDI Syntax for UDDs

> **Note:**
>
> You can use the helper methods `weblogic.jms.extensions.JMSModuleHelper` class `uddMemberName` and `uddMemberJNDIName` APIs to help create UDD CDI names in the correct syntax.

This section provides an example of how to reference a UDD member using `createQueue` or `createTopic` using CDI:

- For an individual member when CDI is not configured, a string defined by:

  `jms-server-name/module-name!jms-server-name@udd-name`

- For an individual member when CDI is configured, a string defined by:

  `jms-server-name/cdi-name`

- A logical UDD is referenced using a string defined by: `module-name!udd-name`.

> **Note:**
>
> When `jms-server-name` is replaced with ".", the API returns the first locally available/started member of the UDQ. A member is considered to be locally available if the JMS client connection is hosted by the same WebLogic Server that currently hosts the member.

Weighted Distributed Destinations

> **Note:**
>
> Weighted distributed destinations are deprecated in Weblogic Server 10.3.4.0. Oracle recommends using Uniform Distributed Destinations.

A weighted distributed destination is a set of individually configured regular destinations that has its own JNDI and CDI name. The logical name of the WDD represents the entire set, and is configured as a JNDI name. There is no option for accessing the logical for a WDD using CDI.

- JNDI Syntax for WDDs
- CDI Syntax for WDDs

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 member logical WDD, see JNDI Syntax for Non distributed Destinations.

CDI Syntax for WDDs

This section provides an example of how to reference a WDD member using the`createQueue` or `createTopic`method with and without using CDI:

- There is no option for accessing a WDD logical name using the `createQueue()` or `createTopic()` methods. A logical WDD must always be referenced using a string defined by the JNDI name of the member. Sometimes it is useful to look up the local individual member using the "." server affinity syntax for non distributed destinations.

- For an individual member when CDI is configured on the member, see CDI Syntax for Non distributed destinations.

D

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:**
>
> This guide includes advanced information for experienced JMS developers. Oracle recommends that you use Message Driven Beans (MDBs) when interacting with Distributed Destinations. The MDB container automatically creates and closes internal consumers across all members of a Distributed Destination as needed. It also handles security, threading, pooling, application life cycle, automatic reconnect, and transaction enlistment. If you cannot use MDBs, then you can use simpler workarounds, such as periodically restarting consumers to rebalance consumers across a distributed destination, or if messaging ordering and performance are not a concern, then enabling the distributed queue forwarding option.

Introduction

A distributed destination (DD) is a group of JMS physical destinations (a group of queues or a group of topics) that is accessed as a single logical destination. Messages are load balanced across members, and clients can failover between member destinations.

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.

Basic JMS

A JMS program sets up message sends in three stages:

1. Clients create a JMS connection into WebLogic using a JMS connection factory.
2. Clients use the connection to create JMS sessions and senders.
3. Clients use the senders to send messages.

In WebLogic JMS, the WebLogic server that the client is connected to is called the client's connection host, and messages always route from the sender, through its connection host, and then on to a destination that's in the same cluster as the connection host. Connections stay pinned to their connection host for the life of the connection.

A WebLogic connection factory can be targeted at one or more WebLogic servers. If a client is running on the same WebLogic server where a connection factory is targeted, then the factory always returns a connection with a connection host that is the same server as the client (the connection is local). On the other hand, if a client is not running on a WebLogic server that is included in its connection factory targets, the factory automatically load balances among the targets and returns a connection to one of them.

When working with a distributed destination, senders should always send to the JNDI name of the DQ or PDT (its "logical name") instead of sending to the JNDI names of the individual members, as this enables automatic load balancing behavior.

Senders to Distributed Queues (DQs) and Partitioned Distributed Topics (PDTs)

The default behavior for a sender to a DQ or PDT is: If there are members that run on the sender's connection host, all sent messages go to one of these local members, otherwise messages move in a round-robin among all members.

To force messages from the same DQ or PDT sender to move in a round-robin among all active members even when local members reside on the sender's connection host, use a custom connection factory with `Server Affinity` set to `false` and `Load Balance` set to `true`.

Senders to Replicated Distributed Topics (RDTs)

Senders to RDTs always load balance once and then pin to a particular member for all messages - this member becomes the "sender host". After a message arrives on the

sender host, the message is automatically replicated to every subscription on every RDT member.

If you want to control the initial load balance decision for the sender host so that it is not biased towards being the same as its connection host, then use a connection factory with `Server Affinity` configured to `false` (default is `true`), and Load Balance configured to `true` (the default).

Using the JMS Destination Availability Helper API

The following sections provide information on how to use the `JMSDestinationAvailabilityHelper` APIs:

Overview

When a consumer is created using the client `javax.jms` API and a DD logical JNDI name is specified, the consumer is load balanced to an active DD member and remains pinned to that member over its lifetime. If new members become active after all consumers were created, then the new members have no consumers.

The `JMSDestinationAvailabilityHelper` APIs provide a way to get notifications when destinations become available or unavailable. These notifications can help ensure that an application creates consumers on all DD members even when there are unavailable members at the time the application is initialized. The same mechanism can also be used to detect availability of other types of destinations (not just WebLogic distributed destinations, but also regular destinations and foreign vendor destinations).

Applications register a notification listener with the helper by specifying JNDI context parameters and the JNDI name of a destination. For DDs, the helper notifies listeners when members become available and unavailable, as they are undeployed, added as a new member, migrated, shut down, or restarted.

Note that MDBs in WebLogic Server internally use this same mechanism for both local MDBs (deployed in the same cluster as a DD) and remote MDBs (deployed in a cluster that is separate from the cluster that hosts the DD). MDBs provide an out-of-the-box solution that achieves the same dynamic adaptability to DD topology changes that the `JMSDestinationAvailabilityHelper` APIs provide.

General Flow

Applications that use the `JMSDestinationAvailabilityHelper` APIs should follow these general steps:

1. Implement the `weblogic.jms.extensions.DestinationAvailableListener` interface to provide behavior as per step 3 below.
2. Register interest with the helper by specifying JNDI context properties (typically just a URL and context factory), the JNDI name of the destination, and a listener instance. Do not specify a URL if the client is running in the same cluster as the DD.

```
import java.util.Hashtable;
import javax.naming.Context;
import weblogic.jms.extensions.JMSDestinationAvailabilityHelper;

Hashtable contextProps = new Hashtable();
contextProps.put(javax.naming.Context.PROVIDER_URL, myURL);
contextProps.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialCon
textFactory");
JMSDestinationAvailabilityHelper dah = JMSDestinationAvailabilityHelper.getIn
stance();

RegistrationHandler rh = dah.register(
   contextProperties,
   destinationJNDIName,
   myDestinationAvailableListener
)
```

3. Handle listener callbacks. Callbacks are single-threaded for each listener instance, so no two callbacks occur concurrently.
   a. `onDestinationsAvailable()`: Typically the first notification. Implementations of this callback usually react by creating zero or more consumers on each given destination, and if this fails, periodically retrying.
   b. `onDestinationsUnavailable()`: This callback is usually used to destroy existing consumers on the destination.
   c. `onFailure()`: This callback is usually used simply to log a failure. The helper continues to retry internally and make subsequent callbacks, but administrators may need to see the failure. The helper makes a best effort to just call the`onFailure()` method once for the same repeated failures.
4. When you are done, unregister interest in a destination by calling the `rh.unregister()` method.

Handling the weblogic.jms.extension.DestinationDetail

As described previously, an `onDestinationsAvailable()` notification indicates that a stand alone destination, foreign destination, or distributed destination member has become available. The notification consists of a list of `DestinationDetail` instances, where key information is obtained by calling the`getDestinationType()`, `getJNDIName()`, `isLocalWLSServer()`, and `isLocalCluster()` on each Detail.

The destination detail helps determine the actions that the caller should take. If the destination is of type `DD_QUEUE`, `REPLICATED_DT`, or `PARTITIONED_DT` then the detail's `getJNDIName()` method returns the JNDI name of a specific DD member and the caller may or may not want to deploy instances of the application consumer on the member. If the destination is of type `PHYSICAL` or `FOREIGN`, then the application treats the destination as a regular destination.

Especially when working with DDs, it is highly recommended that you take advantage of the co-location flags in `DestinationDetail`. You can determine the co-location nature of a destination by calling `isLocalWLSServer()`, and `isLocalCluster()`. See Best Practice for Local Server Consumers.

For more information about APIs and their methods, 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
- URL Handling
- Failure Handling
- JNDI Context Handling
- JMS Connection Handling

When to Register and Unregister

1. Register with `JMSDestinationAvailabilityHelper` at application deployment time. Do not fail the deployment if the helper calls the `onFailure()` callback on your listener (assume it could be an intermittent failure).
2. Unregister with `JMSDestinationAvailabilityHelper` at application undeployment time.

URL Handling

1. If the client is running on the same server or same cluster as the destination, then don't specify a URL when registering with the helper or creating a JNDI context. This ensures that the helper creates a local context.
2. Consider logging a single warning if `isLocalCluster()` or `isLocalServer()` returns `true`, but a URL was specified (as no URL is needed in this case).

Failure Handling

1. Log the errors reported by `onFailure()` notifications, so that the application developer can have a chance to correct possible configuration/application errors. Avoid repeatedly logging the same exception. The helper continues to retry internally and make subsequent callbacks on success or different types of failures, but administrators may need to see the failures. The error may be caused by an application or administrative error such as an incorrect URL, invalid security information, or non-existent destination. It might also be caused by temporary unavailability of the JNDI context host or the destination.
2. When a JMS call throws an exception, or when a JMS connection exception listener reports a connection failure, close the connection. Once all resources have been cleaned up, then periodically attempt to re-initialize all resources. Re-initialization generally involves creating a context, performing JNDI lookups, and then creating a connection, session, and a consumer.
3. Avoid immediately retrying after a failure. Instead periodically retry every few seconds to avoid overloading the server.

JNDI Context Handling

1. In general, avoid creating multiple JNDI initial context instances to the same server or cluster.

   > **Note:**
   >
   > It may be necessary to use additional context instances to work around some security problems, especially in inter-domain scenarios.

2. Call `close()` on a context on undeploy to prevent a memory leak.
3. Call `close()` on a context and re create on any failure (including a lookup failure).

JMS Connection Handling

1. For JMS connections, always register a standard JMS connection "exception listener".
2. On an onException(), close the connection and periodically retry JNDI lookups, recreating a JMS connection, and setting up consumers in another thread.
3. Close connections on undeploy to prevent memory leaks.
4. Instead of sharing a WebLogic Server connection among multiple sessions, consider creating one connection per session. With WebLogic Server, multiple connections allow for better load balancing. There is no performance penalty when working with WebLogic Server, but this might have unexpected overhead with foreign vendors, because some foreign vendors create a TCP/IP socket or a similarly expensive resource for each connection.

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
- Foreign Contexts
- Destination Type Support
- Unavailable Notifications
- Interoperating with WebLogic Server 9.0 and Earlier Distributed Queues
- Interoperating with WebLogic Server 10.3.4.0 and Earlier Distributed Topics
- DestinationDetail Fields

API Availability

The public JMS Destination Availability Helper API is available on AS11gR1PS2 (WebLogic Server version 10.3.3) and later clients and servers.

Foreign Contexts

The context properties that are specified when registering a notification listener with the DA Helper can resolve to any valid JNDI context, including contexts from foreign vendors and older versions of WebLogic Server.

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 `JMSDestinationAvailabilityHelper` API works with any type of destination that can be registered in a JNDI context, including non-distributed destinations and foreign vendor destinations. However, unavailable notifications are only generated for DD members and certain `DestinationDetail` fields apply only to DD members. Unavailable notifications do not apply to foreign destinations.

Unavailable Notifications

Unavailable notifications only apply to DD type destinations (DQ_QUEUE, PARTITIONED_DT, REPLICATED_DT).

Interoperating with WebLogic Server 9.0 and Earlier Distributed Queues

When interoperating with a WebLogic Server 9.0 or later DDs, the DA Helper generates notifications for each individual member of the DD, when working with versions prior to 9.0, the helper only generates a single `DestinationDetail` notification which contains the logical JNDI name for the DD destination and `getDestinationType()` returns `PHYSICAL`.

WebLogic Server 9.0 and earlier DDs are usually treated as a regular destination, and consequently have the same limitations as outlined in Application Design Limitations When Using Replicated Distributed Topics.

Interoperating with WebLogic Server 10.3.4.0 and Earlier Distributed Topics

In releases prior to WebLogic Server 10.3.4, there are no features that enable unrestricted (non-exclusive) client IDs or shared subscriptions.

> **Note:**
>
> For information about how to configure unrestricted client-ids and shared subscriptions, see Configure an Unrestricted ClientID and Configure Shared Subscriptions in *Administering JMS Resources for Oracle WebLogic Server*.

To determine if a destination is a WebLogic 10.3.4.0 topic or later, ensure that the destination type is `PHYSICAL_TOPIC, REPLICATED_DT` or `PARTITIONED_DT` and not `FOREIGN_TOPIC` and that `isAdvancedTopicSupported()` returns `true`. A topic prior to WebLogic Server 10.3.4.0:

- Will never be a `PARTITIONED_DT`.

- `PHYSICAL_TOPIC`s are usually treated as regular topics and are limited to one consumer per subscription.

Automatic attempts to durably subscribe to individual members of WebLogic 10.3.4.0 and earlier DT when a logical DT name is specified are not recommended. Oracle recommends that your applications do not support this option and log an error informing users that need durable subscriptions on a of WebLogic 10.3.4.0 and earlier DT to directly specify the JNDI name of a member instead of specifying the logical DT name.

When subscribing non-durably to a distributed topic prior to WebLogic Server 10.3.4.0, Oracle recommends creating a consumer on any single member JNDI name, or on the logical DR name, and ignoring all other notifications (one subscriber gets all messages sent to the DT and there can be only one consumer thread on the subscription).

DestinationDetail Fields

The behavior of some destination detail fields changes based on the type of destination, the JMS vendor, and, when working WebLogic JMS, the WebLogic Server version. See JMSDestinationAvailabilityHelper in *Java API Reference for Oracle WebLogic Server*.

Security Considerations

The following sections provide information about implementing security using the Java EE and WebLogic Server security models:

- WebLogic Server Security Model
- Passing Credentials Between Threads
- Managing Cross-Domain Security
- Authentication of Users
- Securing Destinations
- Securing Wire Data

WebLogic Server Security Model

WebLogic Server credential propagation is thread based in most cases. The current thread credentials are established by specifying them when creating a JNDI context or application descriptor. These credentials are automatically propagated along with any RMI-based calls between JVMs including WebLogic JMS calls.

Passing Credentials Between Threads

The subject associated with a JNDI context is lost if the context instance is passed to and used in a different thread, which can cause security problems in some multi domain application scenarios. The following sections provide methods on passing credentials:

- Using the Same Thread
- Pass as Anonymous User
- Cache and Reuse a Subject from the Initial Context

Using the Same Thread

If possible, you can avoid the issue by using the same thread to create the context, perform all JMS and JNDI operations, and close the context.

Pass as Anonymous User

Use an anonymous subject if the JMS destination and JNDI resources are not secured. In particular, when interoperating among multiple WebLogic domains, it is usually simplest to force all calls to use an anonymous subject if the JMS destination and JNDI resources are not secured. Non-anonymous credentials are typically only valid for a particular domain, leading to security exceptions if an attempt is made to use them for a different domain.

Cache and Reuse a Subject from the Initial Context

The following code provides an example of how to cache a subject and associate it with another thread using an anonymous user.

```
import java.security.PrivilegedExceptionAction;
import java.security.PrivilegedActionException;

import javax.security.auth.Subject;
import weblogic.security.Security;

 class MyClass {

 // don't make the cached subject public
  private Subject subject;

  MyClass() {
    subject = Security.getCurrentSubject();
  }

   void doSomething() {

 // run some operation as the subject on the original thread
     try {
       Security.runAs(subject,new PrivilegedExceptionAction() {
           public Object run()  throws Exception {
             //  do something;
             return null; // or return some Object
       }});
     } catch (PrivilegedActionException e) {
         // handle exception
     }
  }
}
```

Managing Cross-Domain Security

When using the `JMSDestinationAvailabilityHelper` API in communication between different WebLogic domains, refer the Cross-Domain Security Guidelines.

Authentication of Users

The following sections provide methods to provide the username and password when accessing JMS, which authenticates an application user, and also authorizes an application for JNDI and JMS operations.

- Specifying Credentials for a JNDI Context
- Specifying Credentials for a JMS Connection
- Using Credentials of a Foreign JMS Server JNDI Context
- Using Credentials of a Foreign JMS Server Connection

Specifying Credentials for a JNDI Context

In order to access JMS resources, an application must have access to the JNDI provider. The credentials can be supplied when a application code creates an initial context to the JNDI provider. The thread that establishes the initial context carries the subject, and is therefore used for all sub sequential operations. When an application is running on a WebLogic Server and no server URL and security credentials are provided while creating an initial context, the thread continues to have the same credentials that were on the thread before the initial context was created. When the thread that creates an initial context closes the context, the thread will resume the original security credentials that are on the thread before creating the context.

Specifying Credentials for a JMS Connection

The `ConnectionFactory.createConnection()` call optionally supports a username and password. The credentials that are provided at the connection creation time do not have any affect with respect to security in JMS operations on the connection that is created (This is a WebLogic JMS specific behavior for WebLogic JMS Java clients, with the exception of the .NET client). The credentials are only be used to check, whether or not the user is a valid user in the domain where the connection is created.

Using Credentials of a Foreign JMS Server JNDI Context

Configure the Foreign JMS Server instance with JNDI Properties to gain access to the JNDI provider. The JNDI properties contain the options for setting the security principal and credentials.

Using Credentials of a Foreign JMS Server Connection

The user name and password that can be specified when configuring a Foreign Connection Factory mapping are ignored unless you use an EJB or Servlet resource reference to look up the JMS connection factory. See Improving Performance Through Pooling.

Securing Destinations

WebLogic JMS provides the ability to specify ACLs for destinations. This enables the destination to be secured and only authorized users are allowed to perform operations on that destination. See Java Messaging Service (JMS) Resources in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

Securing Wire Data

When an application must protect JMS data passed on a wire, configure the network to use SSL. See Configuring SSL in *Administering Security for Oracle WebLogic Server*.

Transaction Considerations

WebLogic Server JTA transaction propagation is thread-based. The thread that starts a transaction should be the one that commits or rolls back the transaction. If there is a WebLogic JTA transaction on the current thread when you perform `send` or `receive` operations on a WebLogic JMS destination, then the JMS resources are automatically enlisted with the WebLogic transaction manager, and there is no need to perform your own enlistment.

You only need to do explicit "manual" enlistment when there is a need for WebLogic JMS resources to participate in a foreign or third-party transaction, or there's a need for a non-WebLogic destination to participate in a transaction. Enlisting with a foreign transaction manager (TM) is not directly supported on WebLogic JMS stand-alone clients. EJB and Servlet resource references enable automatic enlistment of non-WebLogic JMS vendors with the WebLogic TM.

Applications should not use transacted sessions if JMS operations are required to participate in a global XA— transaction. Global transactions require use of XA-based connection factories, while local transactions use non-XA based JMS connection factories.

Strategies for Uniform Distributed Queue Consumers

A consumer application can be either running in the same JVM of a WebLogic Server or not, which are called a "server side consumer" and "stand-alone consumer" respectively.

While a JMS UDQ consumer is deployed on a WebLogic Server or cluster, the application can either run on the same cluster/server as the UDQ, or on a different cluster. We call these two different application configurations the local case and the remote case respectively.

> **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*.

For application that cannot use MDBs in their application architecture for some reason, the following guidelines should be followed:

- General Strategies
- Best Practice for Local Server Consumers

General Strategies

In order to for an application to receive all the messages that are sent to a UDQ, the application must make sure that it creates one consumer on each member of the UDQ using the member JNDI name. This requires that applications know the topology of the domains and UDQ configuration, and this is where JMSDestinationAvailabilityHelper can help.

The general strategy is that each deployment instance of a particular application should register with `JMSDestinationAvailabilityHelper`. The listener will receive notifications about member availability.

- Upon receipt of an `onDestinationsAvailable()` notification, the application gets a list of `DestinationDetail` instances for all available members, and then it must create one or more consumer instances using the member JNDI name for each member in the list. For remote consumers, each instance of the application should create a consumer on each member of the UDQ. For local consumers, the application should create a consumer on the local UDQ member only. See Best Practice for Local Server Consumers.
- Upon receipt of an `onDestinationsUnavailable()` notification, the application gets a list of `DestinationDetail` instances for all destinations that becomes unavailable since the last notification. Then for each member destination in the list, the application must find the consumer previously created for the member destination and close it.

Best Practice for Local Server Consumers

An application should be deployed on the same server, group of servers, or cluster that host the UDQ whenever possible. Under this configuration, for best performance, the application should receive messages only from the local members; local members can be determined using the `DestinationDetail isLocalWLSCluster()` call if the servers are in a cluster or the `isLocalWLSServer()` call for individual servers or individual cluster members. This approach yields high performance because all messaging is local (it avoids transferring messages over network calls), and still ensures that all members are serviced by consumers.

In some use cases, the local server optimization network savings does not outweigh the benefit of distributing message processing for unbalanced queue loads across all JVMs in a cluster. This is especially a concern when message backlogs develop unevenly throughout the cluster, and message processing is expensive. In these use cases, the optimization should be avoided in favor of the general strategy model for remote consumers.

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*.

For all clustered and distributed applications that process messages from a UDT, Oracle recommends using product 10.3.4 or later topics in combination with the following settings:

- Set the Client ID Policy to `Unrestricted`. See Configure an Unrestricted ClientID in *Administering JMS Resources for Oracle WebLogic Server*.
- Set Subscription Sharing Policy to `SHARABLE`. See Configure Shared Subscriptions in *Administering JMS Resources for Oracle WebLogic Server*.
- Use the `JMSDestinationAvailabilityHelper` API to get the notification of member availability
- Always create subscribers on the member destinations.

WebLogic JMS has two types of Uniform distributed topics:

- A replicated distributed topic (RDT) has forwarding capability among its members. As a result, each member of a RDT has a copy of all messages that are sent to the RDT.
- A partitioned distributed topic (PDT) does not have forwarding capability among its members. As a result, each member of a PDT has its own copy of all messages that were sent to this particular member. This is a new type of DT introduced in WebLogic Server 10.3.4.0. See Configuring Partitioned Distributed Topics in *Administering JMS Resources for Oracle WebLogic Server*.

The following subsections discuss configuration requirements and programming patterns when using RDTs and PDTs:

- One Copy Per Instance
- One Copy Per Application

One Copy Per Instance

The one copy per instance pattern ensures that each instance gets a copy of each message published to a topic. For example, if each instance is a JVM, then this pattern ensures that each JVM gets a copy of each message sent to the source topic. The following sections provide information on developing design patterns based on one copy per instance:

- General Pattern Design Strategy for One Copy Per Instance
- Best Practice for Local Server Consumers using One Copy Per Instance

General Pattern Design Strategy for One Copy Per Instance

In order for the instances of a distributed application/container to receive messages that are sent to a DT in a one-copy-per-instance manner, each instance must do the following:

1. Choose a base `ClientID` that will be shared by all connections and a durable subscription name that will be shared by all durable subscribers. The subscription name should uniquely identify your application instance. For example, if each instance runs on a differently named WebLogic Server JVM, then the subscription name for each instance could be based on the WebLogic Server name.
2. Create JMS connections and sessions according to standard JMS specifications. The connection's `ClientID` should be set to the base `ClientID` appended by an identifier that is unique for this instance, For example, use the WebLogic Server name or the third-party application server that the application or container is running on. The `ClientIDPolicy` should be set to `Unrestricted`.
3. Set the `SubscriptionSharingPolicy` to `Sharable`.
4. Register with the `JMSDestinationAvailabilityHelper` for membership availability notifications, specifying the JNDI name of the DT.
5. Set an Exception listener.
6. Upon receipt of an `onDestinationsAvailable()` notification, create a subscriber on each newly available destination in the list. If the DT is a replicated DT, the subscriber must use a "`NOT JMS_WL_DDForwarded`" selector or prefix "`(NOT JMS_WL_DDForwarded) AND`" to the existing application provided selector.
7. Upon receipt of an `onDestinationsUnavailable()` notification, close the corresponding `consumer()`.

Best Practice for Local Server Consumers using One Copy Per Instance

An application should be deployed on the same server, group of servers, or cluster that hosts the UDT whenever possible. Under this configuration, the application needs follow the same steps as outlined in General Pattern Design Strategy for One Copy Per Instance except that it creates consumers only on local members. You can use the `JMSDestinationAvailabilityHelper.DestinationDetail.isLocalWLSServer()` call to determine if a member is local.

One Copy Per Application

The one-copy-per application pattern ensures that an application receives one copy of each message sent to a topic, even when the application is clustered across multiple JVMs. For example: If messages "A", "B", and "C" are sent to a topic, the messages are processed once by the application, instead of getting one-copy-per application instance.

The following sections provide information about developing design patterns based on one-copy-per application:

- General Pattern Design Strategy for One Copy Per Application
- Best Practice for Local Server Consumers Using One Copy Per Application

General Pattern Design Strategy for One Copy Per Application

In order for the instances of a distributed application/container to receive messages that are sent to a DT in a one-copy-per-application manner, each instance must do the following:

1. Choose a base `ClientID` for all connections and the durable subscription name for all durable subscribers. The subscription name should uniquely identify your application instance. For example, if each instance runs on a differently named WebLogic Server JVM, the subscription name for each instance could be based on the WebLogic Server name then..
2. Create JMS connections and sessions according to standard JMS specifications. The connection's `ClientID` should be set to the base `ClientID`. The `ClientIDPolicy` should set to `Unrestricted`.
3. Set the `SubscriptionSharingPolicy` to `Sharable`.
4. Register with the `JMSDestinationAvailabilityHelper` for membership availability notifications, specifying the JNDI name of the DT.
5. Set an Exception listener.
6. Upon receipt of an `onDestinationsAvailable()` notification, create a subscriber on each newly available destination in the list. If the DT is a replicated DT, the subscriber needs to use a "`NOT JMS_WL_DDForwarded`" selector or prefix "`(NOT JMS_WL_DDForwarded) AND`" to the existing application provided selector.

Best Practice for Local Server Consumers Using One Copy Per Application

An application should be deployed on the same server, group of servers, or cluster that hosts the UDT whenever possible. Under this configuration, the application must follow the same step outlined in General Pattern Design Strategy for One Copy Per Application except that it creates consumers only on local members. You can use the `JMSDestinationAvailabilityHelper.DestinationDetail.isLocalWLSServer()` call to determine if a member is local.