BEA WebLogic Server®

Programming WebLogic JMS

Version 9.0 BETA
Revised: December 15, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Introduction and Roadmap

2. Understanding WebLogic JMS

3. Best Practices for Application Design

4. Developing a Basic JMS Application

5. Managing Your Applications

6. Using Multicasting with WebLogic Server

7. Using Distributed Destinations

8. Enhanced J2EE Support for Using WebLogic JMS With EJBs and Servlets

9. Using Message Unit-of-Order

10.Using Transactions with WebLogic JMS

11.WebLogic JMS C API

12.Recovering from a WebLogic Server Failure

13.Porting WebLogic JMS Applications

A. Configuration Checklists

B. JDBC Database Utility

C. Deprecated WebLogic JMS Features

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming WebLogic JMS*.

Document Scope and Audience

This document is a resource for software developers who want to develop and configure applications that include WebLogic Server Java Message Service (JMS). It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Server 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. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-3.

It is assumed that the reader is familiar with J2EE 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.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.
- Chapter 2, "Understanding WebLogic JMS," provides an overview of the Java Message Service. It also describes WebLogic JMS components and features.
- Chapter 3, "Best Practices for Application Design," provides design options for WebLogic Server JMS, application behaviors to consider during the design process, and recommended design patterns.
- Chapter 4, "Developing a Basic JMS Application," describes how to develop a WebLogic JMS application.
- Chapter 5, "Managing Your Applications," describes how to programatically manage your JMS applications using value-added WebLogic JMS features.
- Chapter 6, "Using Multicasting with WebLogic Server," describes how to use Multicasting to enable the delivery of messages to a select group of hosts that subsequently forward the messages to subscribers.
- Chapter 7, "Using Distributed Destinations," describes how to use distributed destinations with WebLogic JMS.
- Chapter 8, "Enhanced J2EE Support for Using WebLogic JMS With EJBs and Servlets," describes "best practice" methods that make it easier to use WebLogic JMS in conjunction with J2EE components, like Enterprise Java Beans and Servlets.
- Chapter 9, "Using Message Unit-of-Order," describes how to use Message Unit-of-Order to provide strict message ordering when using WebLogic JMS queues.
- Chapter 10, "Using Transactions with WebLogic JMS," describes how to use transactions with WebLogic JMS.
- Chapter 11, "WebLogic JMS C API," provides information on how to develop C programs that interoperate with WebLogic JMS.

- Chapter 12, "Recovering from a WebLogic Server Failure," describes how to terminate a JMS application gracefully if a server fails and how to migrate JMS data after server failure.
- Chapter 13, "Porting WebLogic JMS Applications," describes how to port your WebLogic JMS applications to a new release of WebLogic Server.
- Appendix A, "Configuration Checklists," provides monitoring checklists for various WebLogic JMS features.
- Appendix B, "JDBC Database Utility," describes how to use the JDBC database utility to generate new JDBC stores and delete existing ones.
- Appendix C, "Deprecated WebLogic JMS Features,"

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:

- *Configuring and Managing WebLogic JMS* for information about configuring and managing JMS resources.
- *Configuring and Managing WebLogic Store-and-Forward Service* for information about the benefits and usage of the Store-and-Forward service with WebLogic JMS.
- Using the WebLogic Persistent Store for information about the benefits and usage of the system-wide WebLogic Persistent Store.
- *Deploying WebLogic Server Applications* is the primary source of information about deploying WebLogic Server applications.
- WebLogic Server Performance and Tuning contains information on monitoring and improving the performance of WebLogic Server applications.

Samples and Tutorials for the JMS Developer

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

BEA 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 J2EE 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 J2EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the *WL_HOME*\samples\domains\medrec directory, where *WL_HOME* is the top-level installation directory for WebLogic Platform.

MedRec includes a service tier comprised primarily 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.

JMS Examples in the WebLogic Server Distribution

WebLogic Server 9.0 optionally installs API code examples in *WL_HOME*\samples\server\examples\src\examples, where *WL_HOME* is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server 9.0 Start menu.

Additional JMS Examples Available for Download

Additional API examples for download at http://dev2dev.bea.com/code/index.jsp. These examples are distributed as ZIP files that you can unzip into an existing WebLogic Server samples directory structure.

You build and run the downloadable examples in the same manner as you would an installed WebLogic Server example. See the download pages of individual examples for more information at http://dev2dev.bea.com/code/index.jsp.

New and Changed JMS Features In This Release

For a comprehensive listing of the new WebLogic JMS feature introduced in release 9.0, see JMS Features, Enhancements, and Deprecations in *Configuring and Managing WebLogic JMS*

CHAPTER 2

Understanding WebLogic JMS

These sections briefly review the different Java Message Service (JMS) concepts and features, and describe how they work with other application objects and WebLogic Server.

It is assumed the reader is familiar with Java programming and JMS 1.1 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. It fully supports the JMS Specification 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 about the destination 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

The following figure illustrates WebLogic JMS messaging.

Figure 2-1 WebLogic JMS Messaging

As illustrated 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.

J2EE Specification

WebLogic Server is compliant with the Sun Microsystems J2EE 1.4 specification.

JMS Specification

WebLogic Server is fully compliant with the JMS 1.1 Specification and can be used in production.

WebLogic JMS Architecture

Major Components

The major components of the WebLogic JMS Server architecture include:

- JMS servers that can host a defined set of destinations (queues or topics) in a JMS module, with which client applications can interact, and any associated persistent storage that reside on a WebLogic Server instance.
- JMS modules contains configuration resources (destinations, connections factories, etc.), and are defined by XML documents that conform to the `weblogic-jmsmd.xsd` schema.
- Client JMS applications that either produce messages to destinations or consume messages from destinations.
- JNDI (Java Naming and Directory Interface), which provides a server *lookup* facility.
- WebLogic persistent storage (file store or JDBC-accessible) for storing persistent message data.

Understanding the Messaging Models

JMS supports two messaging models: point-to-point (PTP) and publish/subscribe (pub/sub). The messaging models are very similar, except for the following differences:

- PTP messaging model enables the delivery of a message to exactly one recipient.
- Pub/sub 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` and the pub/sub class `javax.jms.Topic` both extend the class `javax.jms.Destination`.

Each message model is described in detail in the following sections.

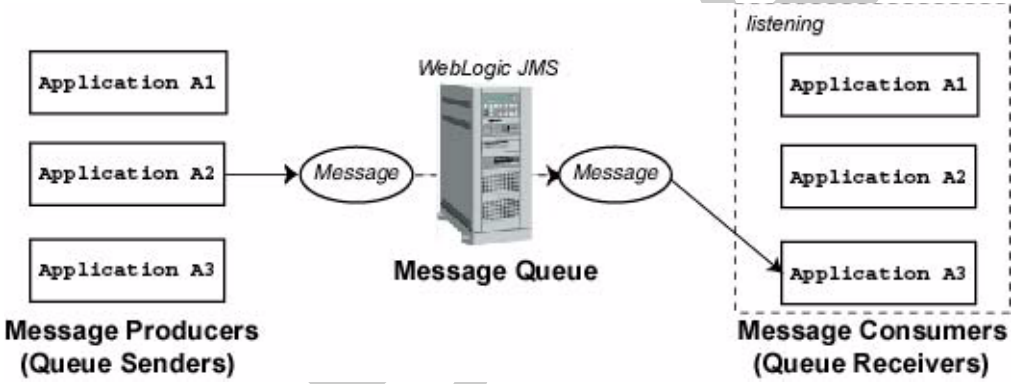
Note: The terms *producer* and *consumer* are used as generic descriptions of applications that send and receive messages, respectively, in either messaging model. For each specific messaging model, however, unique terms specific to that model are used when referring to producers and consumers.

Point-to-Point Messaging

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

The following figure illustrates PTP messaging.

Figure 2-2 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, 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, messages remain in the queue until a queue receiver attaches to the queue.

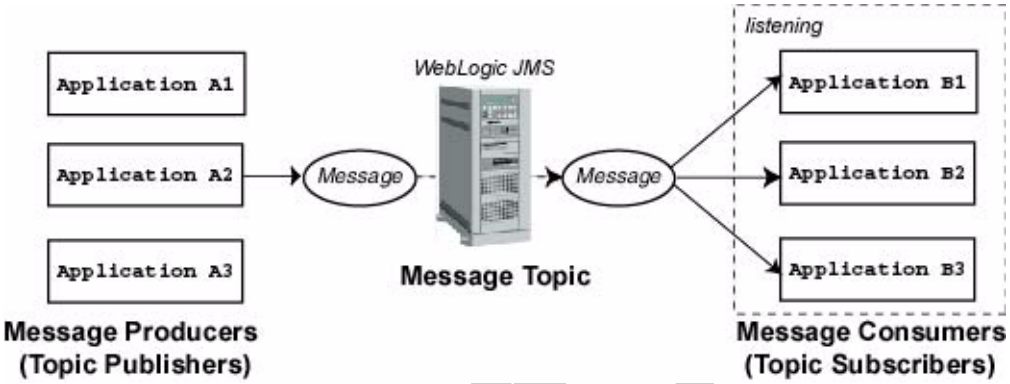
Publish/Subscribe Messaging

The publish/subscribe (pub/sub) messaging model enables an application to send a message to multiple applications. Pub/sub 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.

The following figure illustrates pub/sub messaging.

Figure 2-3 Publish/Subscribe (Pub/Sub) Messaging



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

The Pub/Sub messaging model supports durable subscribers, allowing 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" on page 5-17.

Message Persistence

As per the "Message Delivery Mode" section of the JMS Specification, messages can be specified as persistent or non-persistent:

- A persistent message is guaranteed to be delivered *once-and-only-once*. The message cannot be lost due to a JMS provider failure, but 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 *at-most-once*, 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, 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 Using the WebLogic Persistent Store.

Value-Added Public JMS API Extensions

WebLogic JMS is tightly integrated into the WebLogic Server platform, allowing you to build highly-secure J2EE 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 for WebLogic JMS, see Understanding JMS Resource Configuration in *Configuring and Managing WebLogic JMS*.

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 following table.

Table 2-1 WebLogic JMS Public API Extensions

Class	Function	For more information, see. . .
XMLMessage	Create XML messages	"Step 6a: Create the Message Object (Message Producers)" on page 4-12
WLSession	Define a session exception listener	"Defining a Connection Exception Listener" on page 5-11
WLSession	• Set or display the maximum number of pre-fetched asynchronous messages allowed on a multicast session • Set or display the multicast session overrun policy that is applied when the message maximum is reached	"Dynamically Configuring Multicasting Configuration Attributes" on page 6-4
JMSModuleHelper	Locate JMS runtime MBeans (i.e., monitoring), and manage (locate/create/delete) JMS Module configuration entities (descriptor Beans) in a JMS module	"JMS Module Helper" Javadoc.

Table 2-1 WebLogic JMS Public API Extensions

Class	Function	For more information, see. . .
WLSession	Set a redelivery delay for messages	"Setting a Redelivery Delay for Messages" on page 5-2
WLMessageProducer	Set a message delivery time for producers	"Setting a Delivery Time on Producers" on page 5-6
WLMessage	Set a delivery time for messages	"Setting a Delivery Time on Messages" on page 5-6
Class Schedule	Set a scheduled delivery time for messages	"Setting a Scheduled Time-to-Deliver Override" on page 5-8
ServerSessionPoolFactory	Create server session pools, an optional application server facility described in the JMS specification	"Defining Server Session Pools" on page C-2 **Note:** Session pool configuration objects are deprecated in this release of WebLogic Server. They are not a required part of the J2EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are a required part of J2EE. For more information on designing MDBs, see "Message-Driven EJBs" in *Programming WebLogic Enterprise JavaBeans*.

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

To create a JMS applications, use the `javax.jms` API. The API allows you to create the class objects necessary to connect to the JMS, and send and receive messages. JMS class interfaces are created as subclasses to provide queue- and topic-specific versions of the common parent classes.

The following table lists the JMS classes described in more detail in subsequent sections. For a complete description of all JMS classes, see the `javax.jms` or `weblogic.jms.extensions` Javadoc.

Table 2-2 WebLogic JMS Classes

JMS Class Objects	Description
`ConnectionFactory Object`	Encapsulates connection configuration information. A connection factory is used to create connections. You look up a connection factory using JNDI.
`Connection Object`	Represents an open communication channel to the messaging system. A connection is used to create sessions.
`Session Object`	Defines a serial order for the messages produced and consumed.
`Destination Object`	Identifies a queue or topic, encapsulating the address of a specific provider. Queue and topic destinations manage the messages delivered from the PTP and pub/sub messaging models, respectively.
`MessageProducer and MessageConsumer Objects`	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.
`Message Object`	Encapsulates information to be sent or received.
`ServerSessionPoolFactory Object`[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 Object`[1]	Provides a pool of server sessions that can be used to process messages concurrently for connection consumers.

Table 2-2 WebLogic JMS Classes

JMS Class Objects	Description
`ServerSession Object`[1]	Associates a thread with a JMS session.
`ConnectionConsumer Object`[1]	Specifies a consumer that retrieves server sessions to process messages concurrently.

1. Supports an optional JMS interface for processing multiple messages concurrently.

For information about configuring JMS objects, see "Configuring JMS System Resources" in *Configuring and Managing WebLogic JMS*. The procedure for setting up a JMS application is presented in "Setting Up a JMS Application" on page 4-3.

ConnectionFactory Object

A `ConnectionFactory` object encapsulates connection configuration information, and enables JMS applications to create a `Connection Object`. A connection factory supports concurrent use, enabling multiple threads to access the object simultaneously. You can use the preconfigured 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 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 configure a connection factory if the preconfigured settings of the default factories are not suitable for your application. The main difference between the preconfigured settings for the default connection factories and a user-defined connection factory is the default value for the "XA Connection Factory Enabled" attribute to enable JTA transactions, as shown in the following table.

Table 2-3 XA Transaction(al) 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 Chapter 10, "Using Transactions with WebLogic JMS."

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

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.

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 "Configuring and Deploying Connection Factories" on page 2-10.

Note: For backwards compatibility, WebLogic JMS still supports two deprecated default connection factories. The JNDI names for these factories are: `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory`. For information on migrating to a new default or user-defined connection factory from a deprecated connection factory, refer to "Porting WebLogic JMS Applications" on page 13-1.

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 or the server will not boot.

A system administrator can also establish cluster-wide, transparent access to JMS destinations from any server in the cluster, either by enabling the default connection factories for each server instance in the cluster, or by configuring one or more connection factories and targeting them 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 on JMS clustering, refer to "Configuring Clustered WebLogic JMS Resources" in *Configuring and Managing WebLogic JMS*.

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.

The following table describes the `ConnectionFactory` subclasses.

Table 2-4 ConnectionFactory Subclasses

Subclass. . .	In Messaging Model. . .	Is Used to Create. . .
`QueueConnectionFactory`	PTP	`QueueConnection` to a JMS PTP provider.
`TopicConnectionFactory`	Pub/Sub	`TopicConnection` to a JMS Pub/Sub provider.

To learn how to use the `ConnectionFactory` class within an application, see "Developing a Basic JMS Application" on page 4-1, or the `javax.jms.ConnectionFactory` Javadoc.

Connection Object

A `Connection` object represents an open communication channel between an application and the messaging system, and is used to create a `Session Object` 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 a `ConnectionFactory Object`, 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 may 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" on page 5-13.

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

The following table describes the `Connection` subclasses.

Table 2-5 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 pub/sub provider created by `TopicConnectionFactory`.

To learn how to use the `Connection` class within an application, see "Developing a Basic JMS Application" on page 4-1, or the `javax.jms.Connection` Javadoc.

Session Object

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 an application wants to have a separate thread for producing and consuming messages, the application should create a separate session for each function.

A Session is created by the Connection Object.

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.

The following table describes the Session subclasses.

Table 2-6 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 pub/sub provider. Created by TopicConnection.

To learn how to use the Session class within an application, see "Developing a Basic JMS Application" on page 4-1, or the `javax.jms.Session` and `weblogic.jms.extensions.WLSession` javadocs.

Non-Transacted Session

In a non-transacted session, the application creating the session selects one of the five acknowledge modes defined in the following table.

Table 2-7 Acknowledge Modes Used for Non-Transacted Sessions

Acknowledge Mode	Description
`AUTO_ACKNOWLEDGE`	The `Session` object acknowledges receipt of a message once the 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. Once 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 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.
`DUPS_OK_ACKNOWLEDGE`	The `Session` object acknowledges receipt of a message once 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.

Table 2-7 Acknowledge Modes Used for Non-Transacted Sessions (Continued)

Acknowledge Mode	Description
NO_ACKNOWLEDGE	No acknowledge is required. Messages sent to a NO_ACKNOWLEDGE session are immediately deleted from the server. Messages received in this mode are not recovered, and as a result messages may be lost and/or duplicate message may be delivered if an initial attempt to deliver a message fails. This mode is supported for applications that do not require the quality of service provided by session acknowledge, and that do not want to incur the associated overhead. **Note:** You should avoid using this mode if your application cannot handle lost or duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.
MULTICAST_NO_ACKNOWLEDGE	Multicast mode with no acknowledge required. Messages sent to a MULTICAST_NO_ACKNOWLEDGE session share the same characteristics as NO_ACKNOWLEDGE mode, described previously. This mode is supported for applications that want to support multicasting, and that do not require the quality of service provided by session acknowledge. For more information on multicasting, see "Using Multicasting with WebLogic Server" on page 6-1. **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.

Transacted Session

In a transacted session, only one transaction is active at any given time. Any 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, 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 as 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" on page 10-4.

Destination Object

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 the 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 multiple physical destinations as members of a single distributed destination set within a server cluster. For more information, see "Distributed Destinations" on page 2-16.

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 `Queues` and `Topics` extend `javax.jms.Destination`. The following table describes the `Destination` subclasses.

Table 2-8 Destination Subclasses

Subclass. . .	In 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 pub/sub provider.
`TemporaryTopic`	Pub/sub	JMS pub/sub 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" on page 5-21.

To learn how to use the `Destination` class within an application, see "Developing a Basic JMS Application" on page 4-1, or the `javax.jms.Destination` Javadoc.

Distributed Destinations

Administrators can configure multiple physical destinations as members of a single distributed destination set within a WebLogic Server cluster. Once properly configured, your producers and consumers are able to send and receive to the distributed destination. WebLogic JMS then distributes the messaging load across all available destination members within the distributed destination.

- For more information on using a distributed destination with your applications, see "Using Distributed Destinations" on page 7-1.

MessageProducer and MessageConsumer Objects

A `MessageProducer` object sends messages to a queue or topic. A `MessageConsumer` object 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 Object` 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. The following table describes the `MessageProducer` and `MessageConsumer` subclasses.

Table 2-9 MessageProducer and MessageConsumer Subclasses

Subclass. . .	In Messaging Model. . .	Performs the Following Function. . .
`QueueSender`	PTP	Sends messages for a JMS point-to-point provider.
`QueueReceiver`	PTP	Receives messages for a JMS point-to-point provider, and exists until the JMS connection in which the messages are created is closed.
`TopicPublisher`	Pub/sub	Sends messages for a JMS pub/sub provider.
`TopicSubscriber`	Pub/sub	Receives messages for a JMS pub/sub provider, and exists for the duration of the JMS connection in which the messages are created. Message destinations must be bound explicitly using the appropriate JNDI interface.

The PTP model, as shown in the figure "Point-to-Point (PTP) Messaging" on page 2-4, 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 pub/sub model, as shown in the figure "Publish/Subscribe (Pub/Sub) Messaging" on page 2-5, allows messages to be delivered to multiple topic subscribers. Topic subscribers can be durable or non-durable, as described in "Setting Up Durable Subscriptions" on page 5-17.

An application can use the same JMS connection to both publish and subscribe to a single topic. Because topic messages are 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 Using the Session and Destinations" on page 4-9.

To learn how to use the `MessageProducer` and `MessageConsumer` classes within an application, see "Setting Up a JMS Application" on page 4-3, or the `javax.jms.MessageProducer` and `javax.jms.MessageConsumer` javadocs.

Message Object

A `Message` object encapsulates the information exchanged by applications. This information includes three components: a set of standard header fields, a set of application-specific properties, and a message body. The following sections describe these components.

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" on page 5-21, or to the `javax.jms.Message` Javadoc.

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

Table 2-10 Message Header Fields

Field	Description	Defined by
`JMSCorrelationID`	Specifies one of the following: a WebLogic `JMSMessageID` (described later in this table), an application-specific string, or a `byte[]` array. The `JMSCorrelationID` is used to correlate messages. 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. All `JMSMessageIDs` start with an `ID:` prefix. If you use the `JMSCorrelationID` for some other application-specific string, it *must not* begin with the `ID:` prefix.	Application
`JMSDeliveryMode`	Specifies `PERSISTENT` or `NON_PERSISTENT` messaging. When a persistent message is sent, WebLogic JMS stores it in the JMS file or JDBC database. 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 JMS database. 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. Once a non-persistent message is acknowledged, it will not be redelivered. When a message is sent, this value is ignored. When the message is received, it contains the delivery mode specified by the sending method.	`send()` method

Table 2-10 Message Header Fields (Continued)

Field	Description	Defined by
`JMSDeliveryTime`	Defines the earliest absolute time at which a message can be delivered to a consumer. 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. The application's message producer sets the value of this field when the message is sent. When a message is sent, this value is ignored. When a message is received, its destination value must be equivalent to the value assigned when it was sent.	`send()` method
`JMSExpiration`	Specifies the expiration, or time-to-live value, for a message. 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, `JMSExpiration` 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. All `JMSMessageID`s start with an `ID:` prefix. When a message is sent, this value is ignored. When the message is received, it contains a provider-assigned value.	`send()` method
`JMSPriority`	Specifies the priority level. This field is set before a message is sent. JMS defines ten priority levels, 0 to 9, 0 being the lowest priority. Levels 0-4 indicate gradations of *normal* priority, and level 5-9 indicate gradations of *expedited* priority. When the message is received, it contains the value specified by the method sending the message.	`send()` method

Table 2-10 Message Header Fields (Continued)

Field	Description	Defined by
`JMSRedelivered`	Specifies a flag set when a message is redelivered because no acknowledge was received. This flag is of interest to a receiving application only. 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" on page 4-31.	WebLogic JMS
`JMSReplyTo`	Specifies a queue or topic to which reply messages should be sent. This field is set by the sending application before the message is sent. This feature can be used with the `JMSCorrelationID` header field to coordinate request/response messages. Simply setting the `JMSReplyTo` field does not guarantee a response; it simply *enables* the receiving application to respond. You may set the `JMSReplyTo` to null, which may have a semantic meaning to the receiving application, such as a notification event.	Application

Table 2-10 Message Header Fields (Continued)

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 by the sending application. The JMS specification allows some flexibility with this field in order 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. The following values are valid: boolean, byte, double, float, int, long, short, and String.

Although message property fields may be used for application-specific purposes, JMS provides them primarily for use in message selectors. For more information about message selectors, see "Filtering Messages" on page 5-29.

For information about setting message property fields, see "Setting and Browsing Message Header and Property Fields" on page 5-21, or to the `javax.jms.Message` Javadoc.

Message Body

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

The following table describes the types of messages defined by JMS. All message types extend `javax.jms.Message`, which consists of message headers and properties, but no message body.

Table 2-11 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`.
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.
javax.jms.StreamMessage	Similar to a BytesMessage, except that only Java primitive types are written to or read from the stream.
javax.jms.TextMessage	Single String. The TextMessage can also contain XML content.
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.

For more information, see the `javax.jms.Message` Javadoc. 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 Object

Note: Session pools are now used rarely, as they are not a required part of the J2EE 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 on designing MDBs, see "Message-Driven EJBs" in *Programming WebLogic Enterprise JavaBeans*.

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>`, where <*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" on page C-2, or the `weblogic.jms.extnesions.ServerSessionPoolFactory` Javadoc.

ServerSessionPool Object

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` object obtained through a JNDI lookup.

To learn how to use the server session pool within an application, see "Defining Server Session Pools" on page C-2, or the `javax.jms.ServerSessionPool` Javadoc.

ServerSession Object

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` is created by a `ServerSessionPool Object` object.

To learn how to use the server session within an application, see "Defining Server Session Pools" on page C-2, or the `javax.jms.ServerSession` Javadoc.

ConnectionConsumer Object

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

A `ConnectionConsumer` is created by a `Connection Object` object.

To learn how to use the connection consumers within an application, see "Defining Server Session Pools" on page C-2, or the `javax.jms.ConnectionConsumer` Javadoc.

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

CHAPTER 3

Best Practices for Application Design

These sections discuss design options for WebLogic Server JMS, application behaviors to consider during the design process, and recommended design patterns.

Message Design

This section provides information on how to design messages 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 this cost, to some extent, by having application objects implement `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 `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 even 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.

Message properties

WebLogic JMS message properties are not paged when a message is paged out, thereby continuing to consume memory. They also incur an added serialization cost, both on the client and on the server.

Server-side serialization

WebLogic JMS servers do not incur the cost of serializing Object, Map, Stream, and Bytes messages. Serialization of these message types occurs on the client, but they are treated as simple byte buffers on the server.

Selection

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

Message Compression

Compressing large messages in a JMS application can improve performance. This 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, compression can actually increase its size. The JDK provides built-in compression libraries. For details, see the "java.util.zip" package.

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.

Also, 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.

Topics vs. Queues

Surprisingly, when you are starting to design your application, it is not always immediately obvious whether it would be better to use a Topic or Queue. In general, you should choose 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 would select it.
- There are many subscribers, each with a unique selector.

It is interesting to 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, all previous messages in the subscription are deleted, while if you change a queue selector for consumer, 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 only one consumer.

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.
- 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 via 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.

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. We recommend non-persistent mode because unless synchronous writes are disabled, a persistent QOS almost certainly causes a significant degradation in performance.

Note: Take special care to avoid persisting 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 unintentionally 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:
  - No store configured implies Non-Persistent. (default)
  - Store configured implies no-override.
- Non-durable subscribers only:
  - In WebLogic releases 7.0 and higher, 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 to persist the message.)
- Temporary destinations:
  - Because temporary destinations exist only for the lifetime of their host JMS server, there is no reason to persist their messages. 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 is persisted to ensure that it continues through a server restart, as required by the JMS specification. WebLogic JMS will throw a JMSException if an attempt is made to create a durable subscription on a JMS server with no store configured.

Deferring Acknowledges and Commits

Because sends are generally faster than receives, consider reducing the overhead associated with receives by deferring acknowledgment of messages until several messages have been received and can be acknowledged collectively. If you are using transactions substitute the word "commit" for "acknowledge."

Deferment of acknowledgements is not likely to improve performance for non-durable subscriptions, however, because of 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—provided the wake-up message can be directed solely at the correct listener.

Using AUTO_ACK for Non-Durable Subscribers

In WebLogic Server 7.0 and higher, 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% 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, we recommended 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 6.0 and above provide alternative qualities of service (QOS) extensions that can aid performance.

Using MULTICAST_NO_ACKNOWLEDGE

Non-durable topic subscribers can subscribe to messages using `MULTICAST_NO_ACKNOWLEDGE`. If a topic has such subscribers, 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 only 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 BEA WebLogic Server for non-durable topic subscriptions. The JMS 1.0.2 specification specifically allows non-durable topic messages to be dropped (deleted) if the subscriber is not ready for them. WebLogic JMS actually has a higher QOS for non-durable topic subscriptions by default than the JMS 1.0.2 specification requires.

Using NO_ACKNOWLEDGE

A no-acknowledge delivery mode implies that the server gives messages to consumers, but does not expect acknowledge to be called. Instead, the server pre-acknowledges the message. In this acknowledge mode, calls to recover will not work, as the message is already acknowledged. This mode saves the overhead of an additional network call to 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 `close()` in this scenario, all messages in the asynchronous pipeline are lost.

Asynchronous consumers that use a NO_ACKNOWLEDGE QOS may wish to tune down their message pipeline size in order to reduce the number of lost messages in the event of a crash.

CHAPTER 4

Developing a Basic JMS Application

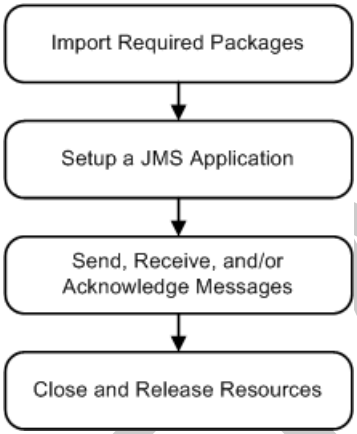
The following sections describe how to develop a basic WebLogic JMS application:

Note: For more information about the JMS classes described in this section, access the JMS Javadoc supplied on the Sun Microsystems' Java Website.

Application Development Flow

When developing a WebLogic JMS application, you must perform the steps identified in the following figure.

Figure 4-1 WebLogic JMS Application Development Flow—Required Steps



In addition to the application development steps defined in the previous figure, you can also optionally perform any of the following steps during your design development:

- Manage connection and session processing
- Create destinations dynamically
- Create durable subscriptions
- Manage message processing by setting and browsing message header and property fields, filtering messages, and/or processing messages concurrently
- Use multicasting
- Use JMS within transactions (described in "Using Transactions with WebLogic JMS" on page 10-1)

Except where noted, all application development steps are described in the following sections.

Importing Required Packages

The following table lists the packages that are commonly used by WebLogic JMS applications.

Table 4-1 WebLogic JMS Packages

Package	Description
javax.jms	Sun Microsystems' JMS API. This package is always used by WebLogic JMS applications.
javax.naming weblogic.jndi	JNDI packages required for server and destination lookups.
javax.transaction.UserTransaction	JTA API required for JTA user transaction support.
weblogic.jms.ServerSessionPoolFactory	Deprecated in WebLogic Server 8.1. See weblogic.jms.extensions.ServerSessionPoolFactory.
weblogic.jms.extensions	WebLogic-specific JMS public API that provides additional classes and methods, as described in "Value-Added Public JMS API Extensions" on page 2-6.

Include the following package `import` statements at the beginning of your program:

```
import javax.jms.*;
import javax.naming.*;
import javax.transaction.*;
```

If you implement a server session pool application, also include the following class on your import list:

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

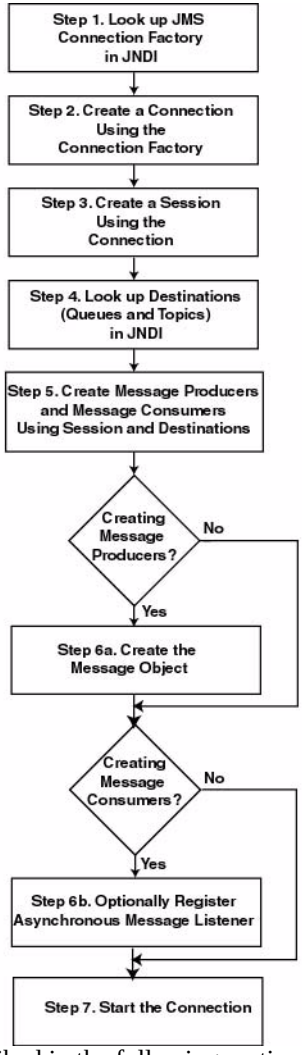
If you want to utilize any of the WebLogic JMS extension classes described in the previous table, also include the following statement on your import list:

```
import weblogic.jms.extensions.*;
```

Setting Up a JMS Application

Before you can send and receive messages, you must set up a JMS application. The following figure illustrates the steps required to set up a JMS application.

Figure 4-2 Setting Up a JMS Application



The setup steps are described in the following sections. Detailed examples of setting up a Point-to-Point (PTP) and Publish/Subscribe (Pub/Sub) application are also provided. The examples are excerpted from the `examples.jms` package provided with WebLogic Server in the *`WL_HOME`*`\samples\server\examples\src\examples\jms` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation.

Before proceeding, ensure that the system administrator responsible for configuring WebLogic Server has configured the required JMS features, including the connection factories, JMS servers, and destinations.

For more information about the JMS classes and methods described in these sections, see "Understanding the JMS API" on page 2-8, or the `javax.jms`, or the `weblogic.jms.extensions` Javadoc.

For information about setting up transacted applications and JTA user transactions, see "Using Transactions with WebLogic JMS" on page 10-1.

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. WebLogic JMS provides two default connection factories that are included as part of the configuration. They can be looked up using the JNDI names, `weblogic.jms.ConnectionFactory` and `weblogic.jms.XAConnectionFactory`, which is configured to enable JTA transactions. The administrator can configure new connection factories during configuration; however, these factories must be uniquely named or the server will not boot.

Once the connection factory has been defined, you can look it up by first establishing a JNDI context (`context`) using the `NamingManager.InitialContext()` method. 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.

Once the context is defined, to look up a connection factory in JNDI, execute one of the following commands, for PTP or Pub/Sub 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 Object" on page 2-9 or the `javax.jms.ConnectionFactory` Javadoc.

Step 2: Create a Connection Using the Connection Factory

You can create a connection for accessing a queue or topic using the `ConnectionFactory` methods described in the following sections.

For more information about the `Connection` class, see "Connection Object" on page 2-11 or the `javax.jms.Connection` Javadoc.

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" on page 4-14.

For more information about the `QueueConnectionFactory` class methods, see the `javax.jms.QueueConnectionFactory` Javadoc. For more information about the `QueueConnection` class, see the `javax.jms.QueueConnection` Javadoc.

Create a Topic Connection

The `TopicConnectionFactory` provides the following two methods for creating 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" on page 4-14.

For more information about the TopicConnectionFactory class methods, see the javax.jms.TopicConnectionFactory Javadoc. For more information about the TopicConnection class, see the javax.jms.TopicConnection Javadoc.

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.

For more information about the Session class, see "Session Object" on page 2-12 or the javax.jms.Session Javadoc.

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, as described in Table 2-7, "Acknowledge Modes Used for Non-Transacted Sessions," on page 2-13. 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. For more information about the QueueSession class, see the javax.jms.QueueSession Javadoc.

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, as described in "Acknowledge Modes Used for Non-Transacted Sessions" on page 2-13. 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. For more information about the `TopicSession` class, see the `javax.jms.TopicSession` Javadoc.

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.

Once the destination has been configured, you can look up a destination by establishing a JNDI context (`context`), which has already been accomplished in "Step 1: Look Up a Connection Factory in JNDI" on page 4-5, 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.

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

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

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

The syntax for the `queueName` and/or `topicName` string is `JMS_Server_Name/Destination_Name` (for example, *`myjmsserver/mydestination`*). To view source code that uses this syntax, refer to the `findqueue()` example in "Dynamically Creating Destinations" on page 5-34.

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 "Dynamically Creating Destinations" on page 5-34.

For more information about these methods, see the `javax.jms.QueueSession` and `javax.jms.TopicSession` Javadoc, respectively.

Once the destination has been defined, you can use the following Queue or Topic method to access the queue or topic name, respectively:

```
public String getQueueName(
) throws JMSException

public String getTopicName(
) throws JMSException
```

To ensure that the queue and topic names are returned in printable format, use the `toString()` method.

For more information about the `Destination` class, see "Destination Object" on page 2-15 or the `javax.jms.Destination` Javadoc.

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.

Step 5: Create Message Producers and Message Consumers Using the Session and Destinations

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. Once 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 write permission.

For more information about the `MessageProducer` and `MessageConsumer` classes, see "MessageProducer and MessageConsumer Objects" on page 2-16, or the `javax.jms.MessageProducer` and `javax.jms.MessageConsumer` Javadocs, respectively.

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" on page 5-29.

If you pass a 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" on page 4-21.

Once the queue sender or receiver has been 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. For more information about the `QueueSender` and

QueueReceiver classes, see the javax.jms.QueueSender and javax.jms.QueueReceiver Javadocs, respectively.

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" on page 5-19. 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 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" on page 5-29.

If you pass a value of null to the createPublisher() method, you create an *anonymous producer*. In this case, you must specify the topic name when sending messages, as described in "Sending Messages" on page 4-21.

An application can have a JMS connection 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.

Once the topic publisher or subscriber has been 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. For more information about the `TopicPublisher` and `TopicSubscriber` classes, see the `javax.jms.TopicPublisher` and `javax.jms.TopicSubscriber` Javadocs, respectively.

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` and `weblogic.jms.extensions.WLSession` Javadocs, respectively. For more information about the `Message` class and its methods, see "Message Object" on page 2-18, or the `javax.jms.Message` Javadoc.

Step 6b: Optionally Register an Asynchronous Message Listener (Message Consumers)

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, which includes an `onMessage()` method.

   Note: For an example of the `onMessage()` method interface, see "Example: Setting Up a PTP Application" on page 4-14.

   If you wish 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 JMS Resources in *Configuring and Managing WebLogic JMS*.

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" on page 5-11.

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

Once a message listener has been 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, a `ConsumerClosedException` is delivered to the session exception listener, if one has been 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" on page 5-11.

The `MessageConsumer` class methods are inherited by the `QueueReceiver` and `TopicSubscriber` classes. For additional information about the `MessageConsumer` class methods, see "MessageProducer and MessageConsumer Objects" on page 2-16 or the `javax.jms.MessageConsumer` Javadoc.

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" on page 5-13 or the `javax.jms.Connection` Javadoc.

Example: Setting Up a PTP Application

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server in the *WL_HOME*`\samples\server\examples\src\examples\jms\queue` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation. The `init()` method shows 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 Object" on page 2-12.

```
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 excerpt 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 `onMessage()` receives a different message type, 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" on page 2-8 or the `javax.jms` Javadoc.

Example: Setting Up a Pub/Sub Application

The following example is excerpted from the `examples.jms.topic.TopicSend` example, provided with WebLogic Server in the *WL_HOME*`\samples\server\examples\src\examples\jms\topic` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation. The `init()` method shows 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 Object" on page 2-12.

```
  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 on 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 PTP Application" on page 4-14.

For more information about the JMS classes used in this example, see "Understanding the JMS API" on page 2-8 or the `javax.jms` Javadoc.

Sending Messages

Once you have set up the JMS application as described in "Setting Up a JMS Application" on page 4-3, you can send messages. To send a message, you must perform the following steps:

1. Create a message object.
2. Define a message.
3. Send the message to a destination.

For more information about the JMS classes for sending messages and the message types, see the `javax.jms.Message` Javadoc. For information about receiving messages, see "Receiving Messages" on page 4-28.

Step 1: Create a Message Object

This step has already been accomplished as part of the client setup procedure, as described in "Step 6a: Create the Message Object (Message Producers)" on page 4-12.

Step 2: Define a Message

This step *may* have been accomplished when setting up an application, as described in "Step 6a: Create the Message Object (Message Producers)" on page 4-12. Whether or not this step has already been accomplished 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 has been specified and you do not wish to change it, you can proceed to step 3.

If a value has not been specified or if you wish 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 the methods used to define messages, see the `javax.jms.Session` Javadoc.

Step 3: Send the Message to a Destination

You can send a message to a destination using a message producer—queue sender (PTP) or topic publisher (Pub/Sub)—and the methods described in the following sections. The `Destination` and `MessageProducer` objects were created when you set up the application, as described in "Setting Up a JMS Application" on page 4-3.

Note: If multiple topic subscribers are defined for the same topic, each subscriber will receive its own local copy of a message. Once 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 `MessageProducer` class, see "MessageProducer and MessageConsumer Objects" on page 2-16 or the `javax.jms.MessageProducer` Javadoc.

Send a Message Using Queue Sender

You can send messages using the following `QueueSender` methods:

```
public void send(
 Message message
) throws JMSException

public void send(
 Message message,
 int deliveryMode,
 int priority,
 long timeToLive
) throws JMSException

public void send(
 Queue queue,
 Message message
) throws JMSException

public void send(
 Queue queue,
 Message message,
 int deliveryMode,
```

```
 int priority,
 long timeToLive
) throws JMSException
```

You must specify a message. You may also specify the queue name (for anonymous message producers), delivery mode (`DeliveryMode.PERSISTENT` or `DeliveryMode.NON_PERSISTENT`), priority (`0-9`), 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:

- Values specified using the message producer's set methods, as described in "Setting Message Producer Attributes" on page 4-25.

Notes: WebLogic JMS also provides the following proprietary attributes, as described in "Setting Message Producer Attributes" on page 4-25:

- `TimeToDeliver` (that is, birth time), which represents the delay before a sent message is made visible on its target destination.
- `RedeliveryLimit`, which determines the number of times a message is redelivered after a recover or rollback.
- `SendTimeout`, which is the maximum time the producer will wait for space when sending a message.

If you define the delivery mode as `PERSISTENT`, you should configure a backing store for the destination.

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.

If the queue sender is an anonymous producer (that is, if when the queue was created, the name was set to null), then you must specify the queue name (using one of the last two methods) to indicate where to deliver messages. For more information about defining anonymous producers, see "Create QueueSenders and QueueReceivers" on page 4-10.

For example, the following code sends a persistent message with a priority of 4 and a time-to-live of one hour:

```
QueueSender.send(message, DeliveryMode.PERSISTENT, 4, 3600000);
```

For additional information about the `QueueSender` class methods, see the `javax.jms.QueueSender` Javadoc.

Send a Message Using TopicPublisher

You can send messages using the following `TopicPublisher` methods:

```
public void publish(
 Message message
) throws JMSException

public void publish(
 Message message,
 int deliveryMode,
 int priority,
 long timeToLive
) throws JMSException

public void publish(
 Topic topic,
 Message message
) throws JMSException

public void publish(
 Topic topic,
 Message message,
 int deliveryMode,
 int priority,
 long timeToLive
) throws JMSException
```


You must provide a message. You may also specify the topic name, delivery mode (`DeliveryMode.PERSISTENT` or `DeliveryMode.NON_PERSISTENT`), priority (`0-9`), 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:

- Values specified using the message producer's set methods, as described in "Setting Message Producer Attributes" on page 4-25.

Notes: WebLogic JMS also provides the following proprietary attributes, as described in "Setting Message Producer Attributes" on page 4-25:

- `TimeToDeliver` (that is, birth time), which represents the delay before a sent message is made visible on its target destination.
- `RedeliveryLimit`, which determines the number of times a message is redelivered after a recover or rollback.
- `SendTimeout`, which is the maximum time the producer will wait for space when sending a message.

If you define the delivery mode as `PERSISTENT`, you should configure a backing store.

Note: If no backing store is configured, then the delivery mode is changed to `NON_PERSISTENT` and no messages are stored.

If the topic publisher is an anonymous producer (that is, if when the topic was created, the name was set to null), then you must specify the topic name (using either of the last two methods) to indicate where to deliver messages. For more information about defining anonymous producers, see "Create TopicPublishers and TopicSubscribers" on page 4-11.

For example, the following code sends a persistent message with a priority of 4 and a time-to-live of one hour:

```
TopicPublisher.publish(message, DeliveryMode.PERSISTENT,
 4,3600000);
```

For more information about the `TopicPublisher` class methods, see the `javax.jms.TopicPublisher` Javadoc.

Setting Message Producer Attributes

As described in the previous section, when sending a message, you can optionally specify the delivery mode, priority, and time-to-live values. If not specified, these attributes are set to the connection factory configuration attributes.

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. The following table 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.

Table 4-2 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`
Time-To-Deliver	`public void setTimeToDeliver( long timeToDeliver ) throws JMSException`	`public long getTimeToDeliver( ) 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 Sun's `javax.jms.MessageProducer` Javadoc or the `weblogic.jms.extensions.WLMessageProducer` Javadoc.

Example: Sending Messages Within a PTP Application

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server in the *WL_HOME*`\samples\server\examples\src\examples\jms\queue` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation. 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.

Example: Sending Messages Within a Pub/Sub Application

The following example is excerpted from the `examples.jms.topic.TopicSend` example, provided with WebLogic Server in the *WL_HOME*`\samples\server\examples\src\examples\jms\topic` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation. 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.

Receiving Messages

Once you have set up the JMS application as described in "Setting Up a JMS Application" on page 4-3, 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, as described in the following sections.

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" on page 4-21.
- Destination sort order set using destination keys.

Once 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. For information about sending messages, see "Sending Messages" on page 4-21.

Receiving Messages Asynchronously

This procedure is described within the context of setting up the application. For more information, see "Step 6b: Optionally Register an Asynchronous Message Listener (Message Consumers)" on page 4-13.

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 have accumulated on an asynchronous consumer, but which have 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 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*.

Behavior of Pipelined Messages

Once 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.

Notes: The Messages Maximum pipeline size setting on the connection factory is not related to the Messages Maximum quota settings on JMS servers and destinations.

Pipelined messages 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 pipelined, the `t3` limit may be exceeded.

Receiving Messages Synchronously

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, 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, 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.

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 *WL_HOME*`\samples\server\examples\src\examples\jms\queue` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation. 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 *WL_HOME*`\samples\server\examples\src\examples\jms\topic` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation. 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.

Recovering Received Messages

Note: This section applies only to non-transacted sessions for which the acknowledge mode is set to `CLIENT_ACKNOWLEDGE`, as described in Table 2-7, "Acknowledge Modes Used for Non-Transacted Sessions," on page 2-13. 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 redelivered in the same order that they were originally delivered, nor to the same queue consumers. For information to guarantee the correct ordering of redelivered messages, see "Ordered Redelivery of Messages" on page 5-4.

Acknowledging Received Messages

Note: This section applies only to non-transacted sessions for which the acknowledge mode is set to `CLIENT_ACKNOWLEDGE`, as described in Table 2-7, "Acknowledge Modes Used for Non-Transacted Sessions," on page 2-13.

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 acknowledge on a message acknowledges all unacknowledged messages received on the session.
- The "Previous" policy specifies that calling 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 have finished 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 impact of the `close()` method for each object, see the appropriate `javax.jms` Javadoc. In addition, for more information about the connection or Session `close()` method, see "Starting, Stopping, and Closing a Connection" on page 5-13 or "Closing a Session" on page 5-15, respectively.

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server in the *WL_HOME*`\samples\server\examples\src\examples\jms\queue` directory, where

WL_HOME is the top-level directory of your WebLogic Platform installation. 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.

CHAPTER 5

Managing Your Applications

The following sections describe how to programatically manage your JMS applications using value-added WebLogic JMS features:

Managing Rolled Back, Recovered, Redelivered, or Expired Messages

The following sections describe how to manage rolled back or recovered messages:

Setting a Redelivery Delay for Messages

You can delay the redelivery of messages when a temporary, external condition prevents an application from properly handling a message. This allows 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, 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 Administration Console.

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.

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 Administration Console.

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. Once 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 to the message destination. If the redelivery limit is configured, but no error destination is configured, then persistent or non-persistent messages are simply 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 Message Producer Attributes" on page 4-25.

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 Administration Console. This setting overrides the redelivery limit set on the message producer.

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 using the Administration Console.

Ordered Redelivery of Messages

As per the JMS Specification, 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 "Designing Message-Driven Beans" in *Programming WebLogic Enterprise JavaBeans*.

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, allowing 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 on the `WLMessageProducer` class, refer to the `weblogic.jms.extensions.WLMessageProducer` Javadoc.

Setting a Delivery Time on Messages

The `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, the `DeliveryTime` can be used to sort messages in a destination or to select messages. For 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 on the `WLMessage` class, refer to the `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 what time-to-deliver is 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

A relative `TimeToDeliverOverride` is a String specified as an integer, and is configurable using the Administration Console.

Setting a Scheduled Time-to-Deliver Override

A scheduled `TimeToDeliverOverride` 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.

Example	Description
`0 0 0,30 * * * *`	Exact next nearest half-hour
`* * 0,30 4-5 * * *`	Anytime in the first minute of the half hours in the 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. Lastly, 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), 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 final example below).
- A value of 1 (one) for the `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 `1` or `last` (not case sensitive) indicates the greatest possible value for a field.
- If a day-of-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, 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 the `TimeToDeliverOverride`. The time returned in milliseconds can be relative or absolute.

For more information on the `WLSession` class, refer to the `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

The following sections describe how to manage 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 be 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.

JMS Metadata	Get Method
Version	`public String getJMSVersion( ) throws JMSException`
Major version	`public int getJMSMajorVersion( ) throws JMSException`
Minor version	`public int getJMSMinorVersion( ) throws JMSException`
Provider name	`public String getJMSProviderName( ) throws JMSException`
Provider version	`public String getProviderVersion( ) throws JMSException`
Provider major version	`public int getProviderMajorVersion( ) throws JMSException`
Provider minor version	`public int getProviderMinorVersion( ) throws JMSException`
JMSX property names	public Enumeration getJMSXPropertyNames( ) throws JMSException

For more information about the `ConnectionMetaData` class, see the `javax.jms.ConnectionMetaData` Javadoc.

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" on page 4-3. 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" on page 10-4.

- 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

The following sections describe how to manage sessions, including:

- 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" on page 4-28.

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 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 (unless such transactions are part of external JTA user transaction). For more information about JTA user transactions, see "Using JTA User Transactions" on page 10-4.
- Does not force an acknowledge 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, the system administrator must select the Allow Close In OnMessage check box when configuring the connection factory.

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" on page 5-17.

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: Before creating a temporary destination (queue or topic), you must use the Administration Console to configure the JMS server to use temporary destinations. This is done by using the JMS server's `Temporary Template` attribute to select a JMS template that is configured in the same domain.

The following sections describe how to create a temporary queue (PTP) or temporary topic (Pub/Sub).

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.

For durable subscriptions, WebLogic JMS stores a message in a persistent file or database until the message has been 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 Pub/Sub 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. For more information on using distributed topics, see "Using Distributed Destinations" on page 7-1.

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
- Defining the Client ID
- Creating Subscribers for a Durable Subscription
- 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 has been 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 JMS Server →Configuration →General tab.

  Note: No two JMS servers can use the same backing store.

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, of course, 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 containing their own client IDs. For more information about configuring a connection factory with a client ID.
- 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
  ```

  You must specify a unique client ID. If you use this alternative approach, 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 `setClientID()` *immediately after* creating their topic connection.

If a client ID is already defined for the connection, an `IllegalStateException` is thrown. If the specified client ID is already defined for another connection, 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, 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 already been defined, use the following Connection method:

```
public String getClientID(
) throws JMSException
```

Note: Support for durable subscriptions is a feature unique to the Pub/Sub 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 Subscribers for a Durable Subscription

You can create subscribers for a durable subscription using the following `TopicSession` methods:

```
public TopicSubscriber createDurableSubscriber(
 Topic topic,
 String name
) throws JMSException

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. 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" on page 5-29. If you do not specify a `messageSelector`, 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 `noLocal` value defaults to `false`.

The durable subscription `name` must be unique per client ID. For information on defining the client ID for the connection, see "Defining the Client ID" on page 5-18.

Only one session can define a subscriber for a particular durable subscription at any given time. Multiple subscribers can access the durable subscription, but not at the same time. Durable subscriptions are stored within the file or database.

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 Administration Console. For information on managing durable subscriptions, see "Managing Durable Subscriptions" on page 5-21.

Modifying Durable Subscriptions

To modify a durable subscription, perform the following steps:

1. Optionally, delete the durable subscription, as described in "Deleting Durable Subscriptions" on page 5-20.

   This step is optional. If 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" on page 5-19 to recreate a durable subscription of the same name, but specifying a different topic name, message selector, or `noLocal` value.

   The durable subscription is recreated based on the new values.

Note: When recreating 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 delete durable subscriptions from the Administration Console.

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 via message selectors. Except for XML message extensions, data in the message body cannot be accessed via message selectors. For example, suppose you use a property field to assign high priority to a message. You can then design a message consumer containing a message selector that accesses this property field and selects only messages of expedited priority. For more information about selectors, see "Filtering Messages" on page 5-29.

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. Once a message is received, its header field values can be modified.

For a description of the standard messages header fields, see "Message Header Fields" on page 2-18.

The following table 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.

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
JMSDeliveryMode[1]	public void setJMSDeliveryMode(int deliveryMode) throws JMSException	public int getJMSDeliveryMode( ) throws JMSException
JMSDeliveryTime[1]	public void setJMSDeliveryTime(long deliveryTime) throws JMSException	public long getJMSDeliveryTime( ) throws JMSException
JMSDeliveryMode[1]	public void setJMSDeliveryMode(int deliveryMode) throws JMSException	public int getJMSDeliveryMode( ) throws JMSException

Header Field	Set Method	Get Method
JMSMessageID[1]	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
JMSPriority[1]	public void setJMSPriority(int priority) throws JMSException	public int getJMSPriority( ) throws JMSException
JMSRedelivered[1]	public void setJMSRedelivered(boolean redelivered) throws JMSException	public boolean getJMSRedelivered( ) throws JMSException
JMSRedeliveryLimit[1]	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

Header Field	Set Method	Get Method
JMSTimeStamp[1]	public void setJMSTimeStamp(long timestamp) throws JMSException	public long getJMSTimeStamp( ) throws JMSException
JMSType	public void setJMSType(String type) throws JMSException	public String getJMSType( ) throws JMSException

1. The corresponding `set()` method has no impact on the message header field when the `send()` method is executed. If set, this header field value will be overridden during the `send()` operation.

The `examples.jms.sender.SenderServlet` example, provided with WebLogic Server in the *WL_HOME*`\samples\server\examples\src\examples\jms\sender` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation, 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" on page 5-12.

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 5-1 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`

Table 5-1 Message Property Set and Get Methods for Data Types (Continued)

Data Type	Set Method	Get Method
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 previous table, based on the property field data type.

The following table is a conversion chart for message properties. It allows you to identify the type that can be read based on the type that has been written.

Table 5-2 Message Property Conversion Chart

Property Written As. . .	Can Be Read As. . .							
	boolean	byte	double	float	int	long	short	String
boolean	X							X
byte		X			X	X	X	X
double			X					X

Table 5-2 Message Property Conversion Chart (Continued)

Property Written As. . .	Can Be Read As. . .							
	boolean	byte	double	float	int	long	short	String
float			X	X				X
int					X	X		X
long						X		X
Object	X	X	X	X	X	X	X	X
short					X	X	X	X
String	X	X	X	X	X	X	X	X

You can test whether or not a property value has been 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" on page 2-23 or the `javax.jms.Message` Javadoc.

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 wish to browse. You may also specify a message selector to filter messages that you are browsing. Message selectors are described in more detail in "Filtering Messages" on page 5-29.

Once you have defined 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 *WL_HOME*`\samples\server\examples\src\examples\jms\queue` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation, shows how to access the header fields of received messages.

For example, the following code line is an excerpt from the `QueueBrowser` example and creates the `QueueBrowser` object:

```
qbrowser = qsession.createBrowser(queue);
```

The following provides 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" on page 4-32.

For more information about the `QueueBrowser` class, see the `javax.jms.QueueBrowser` Javadoc.

Filtering Messages

In many cases, an application does not need to be notified of every message that is delivered to it. Message selectors can be used to filter unwanted messages, and subsequently improve performance by minimizing their impact 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 Using the Session and Destinations" on page 4-9.

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" on page 5-21 and "Setting Message Property Fields" on page 5-24, 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.

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)
```

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 excerpt:

```
<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 quotes and spaces. Note the use of single quotes 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"

where *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 exact 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.

Once subscribers have registered 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 illustrated 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);
```


Notes:

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, as well as 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 no need for the enhancement anyway, since the message selection occurs on the client side rather than the server side.

Using the JMS Module Helper Methods to Manage Your Applications

The JMS Module Helper class contains methods for locating JMS runtime MBeans (that is, monitoring), as well as methods to manage (locate/create/delete) JMS Module configuration entities (Descriptor Beans) in a given JMS module.

The following sections provide additional information on using some JMS Module Helper methods.

Dynamically Creating Destinations

You can create destinations dynamically using:

- `weblogic.jms.extensions.JMSModuleHelper` class methods
- Temporary destinations

The associated procedures for creating dynamic destinations are described in the following sections.

Using the JMSModuleHelper Class Methods

You can dynamically submit an asynchronous request to create a queue or topic, respectively, using the following `JMSModuleHelper` methods available in `weblogic.jms.extensions`:

```
static public void createPermanentQueueAsync(
 Context ctx,
 String jmsServerName,
 String queueName,
 String jndiName
) throws JMSException
```

```
static public void createPermanentTopicAsync(
 Context ctx,
 String jmsServerName,
 String topicName,
 String jndiName
) throws JMSException
```

You must specify the JNDI initial context, name of the JMS server to be associated with the destination, name of the destination (queue or topic), and name used to look up the destination within the JNDI namespace.

Each method updates the following:

- Configuration file associated with the specified domain to include the dynamically created destination
- JNDI namespace to advertise the destination

Note: Either 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 once it becomes available. For example:

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

For more information on the `JMSModuleHelper` class, refer to the `weblogic.jms.extensions.JMSModuleHelper` Javadoc.

Dynamically Deleting Destinations

You can dynamically delete JMS destinations (queue or topic) using any of the following methods:

- `weblogic.jms.extensions.JMSModuleHelper` class method
- Administration console
- User-defined JMX application

The JMS server removes the deleted destination in real time, therefore, it's not necessary to redeploy the JMS server for the deletion to take effect. The associated procedures for dynamically deleting destinations are described in the following sections.

Preconditions for Deleting Destinations

In order to successfully delete a destination, the following preconditions must be met:

- The destination must not be a member of a distributed destination. For more information, see "Using Distributed Destinations" on page 7-1.
- The destination must not be the error destination for some other destination. For more information, see "Configuring an Error Destination for Undelivered Messages" on page 5-4.

If either of these preconditions cannot be met, then the deletion will not be allowed.

Using the JMSModuleHelper Class Methods

You can dynamically submit a request to delete a destination (queue or topic), using the following `JMSModuleHelper` methods available in `weblogic.jms.extensions`:

```
static public void deletePermanentQueue(
  Context ctx,
  String jmsServerName,
  String queueName
 ) throws ConfigurationException
```

```
static public void deletePermanentTopic(
  Context ctx,
  String jmsServerName,
  String topicName
 ) throws ConfigurationException
```

You must specify the JNDI initial context, name of the JMS server to be associated with the destination, and the name of the destination (queue or topic).

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 — once 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` 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 `nextElement()` will return a message.
  - Subsequent calls to `nextElement()` will throw a `NoSuchElementException`.
  - Calls to `hasMoreElements()` made before the first call to `nextElement()` will return true.
  - Calls to `hasMoreElements()` made after the first call to `nextElement()` will return false.

  If a given enumeration has never been called, or the last call before the close was to `nextElement()`, or the last call before the close was to `hasMoreElements()` and the value returned was false, then the following behaviors apply:
  - Calls to `hasMoreElements()` will return false.
  - Calls to `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`. For more information on using blocking send operations.
- 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, since the destination is deleted, the net result of all operations (rollback, commit, etc.) is the deletion of the associated messages.

Message Timestamps for Troubleshooting Deleted Destinations

If a destination with persistent messages is deleted and then immediately recreated while the JMS server is not running, the JMS server will compare the version number of the destination (using the `CreationTime` field in the configuration `config.xml` file) and 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 recreated destination, and when the JMS server is rebooted, the left over persistent messages are simply discarded.

However, if a persistent message somehow has a version number that is *newer* than the version number in the `config.xml` for the recreated destination, then either the system clock was rolled back when the destination was deleted and recreated (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/or received in a transaction, as well as unacknowledged non-transactional messages received by a client.

Sending XML Messages

Note: This release does not support streaming. Only text and DOM representations of XML documents are supported.

Previous releases of the WebLogic Server JMS API only provided messaging of XML documents using the `String` type. For this release, the WebLogic Server JMS API also provides native support for the Document Object Model (DOM) to send XML messages.

The following sections provide information on WebLogic JMS API extensions that provide enhanced support for XML messages.

- "WebLogic XML APIs" on page 5-40
- Using a String Representation

- Using a DOM Representation

WebLogic XML APIs

The section provides information on the 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 `toString` on the `StringWriter` and pass it into `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 `XMLMessage.setDocument`.
3. Publish the message.

CHAPTER 6

Using Multicasting with WebLogic Server

Multicasting enables the delivery of messages to a select group of hosts that subsequently forward the messages to subscribers. The following sections provide information on the benefits, limitations, and configuration of using mutlticasing with WebLogic Server:

Benefits of using Multicasing

The benefits of multicasting include:

- Near real-time delivery of messages to host group.
- High scalability due to the reduction in the amount of resources required by the JMS server to deliver messages to subscribers.

Limitations of using Multicasting

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.

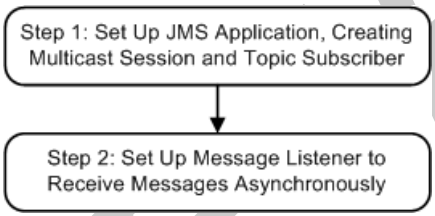
- For interoperability with different versions of WebLogic Server, clients cannot have an earlier release of WebLogic Server installed than the host. They must all have at least the same version or higher.

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 simply 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.

Configuring Multicasting for WebLogic Server

The following figure illustrates the steps required to set up multicasting.

Figure 6-1 Setting Up Multicasting



Note: Multicasting is only supported for the Pub/Sub messaging model, and only for non-durable subscribers.

Monitoring statistics are not provided for multicast sessions or consumers.

Prerequesites 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" on page 6-4.
- 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 TTL" on page 6-5.

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.

The multicast configuration attributes are also summarized in Appendix A, "Configuration Checklists."

Step 1: Set Up the JMS Application, Creating Multicast Session and Topic Subscriber

Set up the JMS application as described in "Setting Up a JMS Application" on page 4-3. However, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-7, 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 Pub/Sub 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 illustrates how to create a multicast session for the Pub/Sub messaging model.

```
tsession = tcon.createTopicSession(
 false,
 WLSession.MULTICAST_NO_ACKNOWLEDGE
);
```

Note: On the client side, each multicasting session requires one dedicated thread to retrieve messages off the socket. Therefore, you should increase the JMS client-side thread pool size to adjust for this. For more information on adjusting the thread pool size, see the "Tuning Thread Pools and EJB Pools" section in the "WebLogic JMS Performance Guide" white paper, at `http://dev2dev.bea.com/resourcelibrary/whitepapers/index.jsp#server`, which discusses tuning JMS client-side thread pools.

In addition, create a topic subscriber, as described in "Create TopicPublishers and TopicSubscribers" on page 4-11.

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, a `JMSException` is thrown.

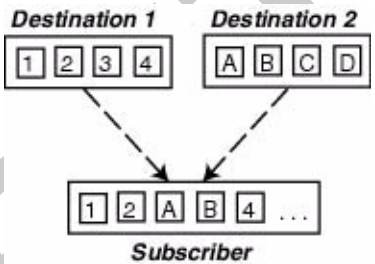
Set up the message listener for the topic subscriber, as described in "Receiving Messages Asynchronously" on page 4-28.

For example, the following code illustrates 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, a `SequenceGapException` will be delivered to the `ExceptionListener` for the session(s) present. If a skipped message is subsequently delivered, it will be discarded. For example, in the following figure, the subscriber is receiving messages from two destinations simultaneously.

Figure 6-2 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:

- Messages 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 messages maximum is reached.

If the messages 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.

The following table lists the `Session` set and get methods for each dynamically configurable attribute.

Table 6-1 Message Producer Set and Get Methods

Attribute	Set Method	Get Method
Messages 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 on these multicast configuration attributes, see "JMS Topic --> Configuration --> Multicast" in the *Administration Console Online Help*.

Example: Multicast TTL

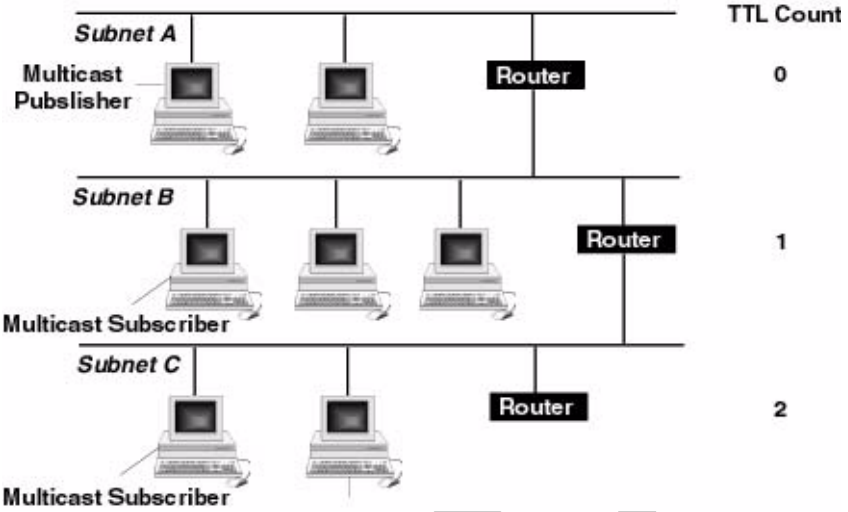
Note: The following example is a very simplified illustration of how the Multicast TTL (time-to-live) destination configuration attribute impacts the delivery of messages across routers. It is strongly advised that you seek the assistance of your network administrator when configuring the multicast TTL attribute to ensure that the appropriate value is set.

The Multicast TTL is independent of the message time-to-live.

The following example illustrates how the Multicast TTL destination configuration attribute impacts the delivery of messages across routers.

Consider the following network diagram.

Figure 6-3 Multicast TTL Example



In the figure, the network consists of three subnets: Subnet A containing the multicast publisher, and Subnets B and C each containing 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.

CHAPTER 7

Using Distributed Destinations

A *distributed* destination is a set of physical destinations (queues or topics) that are called under a single JNDI name so they appear to be a single, logical destination to a client, when the members of the set are actually distributed across multiple servers within a cluster, with each destination member belonging to a separate JMS server.

By enabling you to configure multiple physical queues and topics as members of a distributed destination, WebLogic JMS supports high availability and load balancing of physical destinations 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.

To facilitate the configuration process, the Administration Console has an "Auto Deploy" feature that allows you to easily configure distributed destinations, as follows:

- Create a Distributed Destination and Automatically Create Members — for new implementations of WebLogic JMS with no physical destinations or existing configurations of WebLogic JMS that do not require previously configured destinations to be part of a distributed destination.
- Create a Distributed Destination and Manually Add Existing Physical Destinations as Members — for existing implementations of WebLogic JMS that require previously configured destinations to be members of a distributed destination.

The following sections explain how to use distributed destinations with your JMS applications:

Accessing Distributed Destinations

A distributed destination is actually 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. It implements the `javax.jms.Destination` interface, and can be used to create producers, consumers, and browsers.

Because a distributed destination can be served by multiple WebLogic Servers within a cluster, when creating a reference to a distributed destination by using one of the `createQueue()` or `createTopic()` methods, the name supplied is simply the name of the `JMSDistributedQueueMBean` or `JMSDistributedTopicMBean` configuration MBean name. No JMS server name or separating forward slash (`/`) is required.

For example, the following code illustrates how to look up a distributed destination topic member:

```
topic = myTopicSession.createTopic("myDistributedTopic");
```

Note: When calling the `createQueue()` or `createTopic()` methods, any string containing a forward slash (`/`), is assumed to be the name of a distributed destination member—not a physical destination. If no such destination member exists, then the call will fail with an `InvalidDestinationException`.

Looking Up 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 exactly 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.

Note: Queue members can forward messages to other queue members by configuring the Forward Delay attribute in the 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.

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 in any way. 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. Since 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 on the load-balancing heuristics for distributed queues with zero consumers, see "Load Balancing Heuristics" on page 7-9.

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 won't be able to recreate the queue receiver and will have to try again later.

Note: For information on the load-balancing heuristics for distributed queues with zero consumers, see "Load Balancing Heuristics" on page 7-9.

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.

Looking Up Distributed Topics

A distributed topic is a set of physical JMS topic members. As such, 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 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 allows 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 which 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" on page 7-6.

Deploying Message-Driven Beans on a Distributed Topic

When an MDB is deployed on a distributed topic and is targeted to a WebLogic Server instance in a cluster that is hosting two members of the distributed topic on a JMS server, the MDB gets

deployed on both the members of the distributed topic. This occurs because MDBs are pinned to a distributed topic member's destination name.

Therefore, you will receive [*number of messages sent*] * *[number of distributed topic members*] more messages per MDB, depending on how may distributed topic members are deployed on a WebLogic Server instance. For example, if a JMS server contains two distributed topic members, then two MDBs are deployed, one for each member, so you will receive twice as many messages.

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, as follows:

- If one or more of the distributed topic members is not reachable, and the message being sent is non-persistent, then the message is sent only to the available topic members.
- 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 "Load Balancing Messages Across a Distributed Destination" on page 7-7.

- 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 once 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. As such, 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 non-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.

Upon receiving such 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.

Accessing Distributed Destination Members

In order to access a destination member within a distributed destination, you must look up the destination member using the configured JNDI name, or supply the JMS server name and the `JMSQueueMBean` or `JMSTopicMBean` configuration MBean name, separated by a forward slash (`/`), to one of the `createQueue()` or `createTopic()` methods.

For example, the following code illustrates how to look up a particular member of a distributed queue (`myQueue`), on a JMS server (`myServer`):

```
queue = myQueueSession.createQueue("myServer/myQueue");
```

Note: When calling the `createQueue()` or `createTopic()` methods, any string containing a forward slash (`/`), is assumed to be the name of a distributed destination member—not a destination. If no such destination member exists, then the call will fail with an `InvalidDestinationException`.

Load Balancing Messages Across a Distributed Destination

By using distributed destinations, WebLogic JMS can spread or balance the messaging load across multiple physical destinations, which can result in better use of resources and improved response times. The WebLogic JMS load-balancing algorithm determines the physical destinations that messages are sent to, as well as the physical destinations that consumers are assigned to.

Load Balancing Options

WebLogic JMS supports two different algorithms for balancing the message load across multiple physical destinations within a given distributed destination set. You select one of these load balancing options when configuring a distributed topic or queue on the Administration Console.

- Round-Robin Distribution
- Random Distribution

Round-Robin Distribution

In the round-robin algorithm, WebLogic JMS maintains an ordering of physical destinations within the distributed destination. The messaging load is distributed across the physical destinations one at a time in the order that they are defined in the WebLogic Server configuration (`config.xml`) file. Each WebLogic Server maintains an identical ordering, but may be at a different point within the ordering. Multiple threads of execution within a single server using a given distributed destination affect each other with respect to which physical destination a member is assigned to each time they produce a message. Round-robin is the default algorithm and doesn't need to be configured.

If weights are assigned to any of the physical destinations in the set for a given distributed destination, then those physical destinations appear multiple times in the ordering. For instance, if the weights of destinations A, B and C are 2, 5, and 3 respectively, then the ordering will be A, B, C, A, B, C, B, C, B, B. That is, a number of passes are made through the basic ordering (A, B, C). The number of passes is equal to the highest weight of the destinations within the set. On each pass, only those destinations with a weight that is greater than or equal to the ordinal value of the

pass are included in the ordering. Following this logic, this example would produce the following results:

- A is dropped from the ordering after two passes.
- C is dropped after three passes.
- B is the only one remaining on the fourth and fifth passes.

Random Distribution

The random distribution algorithm uses the weight assigned to the physical destinations to compute a weighted distribution for the set of physical destinations. The messaging load is distributed across the physical destinations by pseudo-randomly accessing the distribution. In the short run, the load will not be directly proportional to the weight. In the long run, the distribution will approach the limit of the distribution. A pure random distribution can be achieved by setting all the weights to the same value, which is typically 1.

Adding or removing a member (either administratively or as a result of a WebLogic Server shutdown/restart event) requires a recomputation of the distribution. Such events should be infrequent however, and the computation is generally simple, running in O(n) time.

Consumer Load Balancing

When an application creates a consumer, it must provide a destination. If that destination represents a distributed destination, then WebLogic JMS must find a physical destination that consumer will receive messages from. The choice of which destination member to use is made by using one of the load-balancing algorithms described in "Load Balancing Options" on page 7-7. The choice is made only once: when the consumer is created. From that point on, the consumer gets messages from that member only.

Producer Load Balancing

When a producer sends a message, WebLogic JMS looks at the destination where the message is being sent. If the destination is a distributed destination, WebLogic JMS makes a decision as to where the message will be sent. That is, the producer will send to one of the destination members according to one of the load-balancing algorithms described in "Load Balancing Options" on page 7-7.

The producer makes such a decision each time it sends a message. However, there is no compromise of ordering guarantees between a consumer and producer, because consumers are load balanced once, and are then pinned to a single destination member.

Note: If a producer attempts to send a persistent message to a distributed destination, 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 persistent store, then the message will still be sent to one of the members according to the selected load-balancing algorithm. bucolic

Load Balancing Heuristics

In addition to the algorithms described in "Load Balancing Options" on page 7-7, WebLogic JMS uses the following heuristics when choosing an instance of a destination.

- Transaction Affinity
- Server Affinity
- Queues with Zero Consumers

Transaction Affinity

When producing multiple messages within a transacted session, an effort is made to send all messages produced to the same WebLogic Server. Specifically, if a session sends multiple messages to a single distributed destination, then all of the messages are routed to the same physical destination. If a session sends multiple messages to multiple different distributed destinations, an effort is made to choose a set of physical destinations served by the same WebLogic Server.

Server Affinity

When the Server Affinity option is enabled for distributed destinations, then before a WebLogic Server instance attempts to load balance consumers or producers across all the members of a distributed destination in a domain, it will first attempt to load balance across any local members that are running on the same WebLogic Server instance.

Note: The Server Affinity Enabled attribute does not affect queue browsers. Therefore, a queue browser created on a distributed queue can be pinned to a remote distributed queue member even when Server Affinity is enabled.

Queues with Zero Consumers

When load balancing consumers across multiple remote physical queues, if one or more of the queues have zero consumers, then those queues alone are considered for balancing the load. Once all the physical queues in the set have at least one consumer, the standard algorithms apply.

In addition, when producers are sending messages, queues with zero consumers are not considered for message production, unless all instances of the given queue have zero consumers.

Defeating Load Balancing

Applications can defeat load balancing by directly accessing the individual physical destinations. That is, if the physical destination has no JNDI name, it can still be referenced using the `createQueue()` or `createTopic()` methods.

- JNDI Lookup
- CreateQueue() and CreateTopic()
- Connection Factories

JNDI Lookup

If a physical destination has a JNDI name, then it can be looked up using JNDI. The returned destination can then be used to create a consumer or receiver.

CreateQueue() and CreateTopic()

An application can also obtain a reference to a topic or queue using the `createQueue()` and `createTopic()` methods. When using these methods, the application must supply a vendor-specific string identifying the destination that they want a reference to. The vendor-specific string for WebLogic JMS is of the form *server/destination*, where “server” is the name of a JMS server and “destination” is the name of a queue or topic on that JMS server.

Connection Factories

Applications that use distributed destinations to distribute or balance their producers and consumers across multiple physical destinations, but do not want to make a load balancing decision each time a message is produced, can use a connection factory with the Load Balancing Enabled attribute disabled (i.e., set to False).

How Distributed Destination Load Balancing Is Affected When Using the "Server Affinity Enabled" Attribute

The following table explains how the setting of the JMS connection factory's Server Affinity Enabled attribute affects the load balancing preferences for distributed destination members. The order of preference depends on the type of operation and whether or not durable subscriptions or persistent messages are involved.

The Server Affinity Enabled attribute for distributed destinations is different from the server affinity provided by the Default Load Algorithm attribute in the `ClusterMBean`, which is also used by the JMS connection factory to create initial context affinity for client connections.

Table 7-1 Server Affinity Load Balancing Preferences

When the operation is...	and Server Affinity Enabled is...	then load balancing preference is given to a...
• `createReceiver()` for queues • `createDurableSubscriber()` for topics	True	1. local member without a consumer 2. local member 3. remote member without a consumer 4. remote member
`createReceiver()` for queues	False	1. member without a consumer 2. member
`createSubscriber()` for topics (**Note:** non-durable subscribers)	True or False	1. local member without a consumer 2. local member 3. local proxy topic
• `createSender()` for queues • `createPublisher()` for topics	True or False	There is no separate machinery for load balancing a JMS producer creation. JMS producers are created on the server on which your JMS connection is load balanced or pinned. For more information about load balancing JMS connections created via a connection factory, refer to the "Load Balancing for EJBs and RMI Objects" and "Initial Context Affinity and Server Affinity for Client Connections" sections in *Using WebLogic Server Clusters*.

Table 7-1 Server Affinity Load Balancing Preferences

When the operation is...	and Server Affinity Enabled is...	then load balancing preference is given to a...
For persistent messages using `QueueSender.send()`	True	1. local member with a consumer and a store 2. remote member with a consumer and a store 3. local member with a store 4. remote member with a store 5. local member with a consumer 6. remote member with a consumer 7. local member 8. remote member
For persistent messages using `QueueSender.send()`	False	1. member with a consumer and a store 2. member with a store 3. member with a consumer 4. member
For non-persistent messages using `QueueSender.send()`	True	1. local member with a consumer 2. remote member with a consumer 3. local member 4. remote member
For non-persistent messages: • `QueueSender.send()` • `TopicPublish.publish()`	False	1. member with a consumer 2. member
`createConnectionConsumer()` for session pool queues and topics	True or False	1. local member *only* **Note:** Session pools are now used rarely, as they are not a required part of the J2EE 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.

Distributed Destination Migration

For clustered JMS implementations that take advantage of the Service Migration feature, a JMS server and its distributed destination members can be manually migrated to another WebLogic Server instance within the cluster. Service migrations can take place due to scheduled system maintenance, as well as in response to a server failure within the cluster.

However, the target WebLogic Server may already be hosting a JMS server with all of its physical destinations. This can lead to situations where the same WebLogic Server instance hosts two physical destinations for a single distributed destination. This is permissible in the short term, since a WebLogic Server instance can host multiple physical destinations for that distributed destination. However, load balancing in this situation is less effective.

In such a situation, each JMS server on a target WebLogic Server instance operates independently. This is necessary to avoid merging of the two destination instances, and/or disabling of one instance, which can make some messages unavailable for a prolonged period of time. The long-term intent, however, is to eventually re-migrate the migrated JMS server to yet another WebLogic Server instance in the cluster.

For more information about the configuring JMS migratable targets, see "Configuring JMS Migratable Targets" in *Configuring and Managing WebLogic JMS*.

Distributed Destination Failover

If the server instance that is hosting the JMS connections for the JMS producers and JMS consumers should fail, then all the producers and consumers using these connections are closed and are *not* re-created on another server instance in the cluster. Furthermore, if a server instance that is hosting a JMS destination should fail, then all the JMS consumers for that destination are closed and not re-created on another server instance in the cluster.

If the distributed queue member on which a queue producer is created should fail, yet the WebLogic Server instance where the producer's JMS connection resides is still running, the producer remains alive and WebLogic JMS will fail it over to another distributed queue member, irrespective of whether the Load Balancing option is enabled.

For more information about procedures for recovering from a WebLogic Server failure, see "Recovering from a WebLogic Server Failure."

CHAPTER 8

Enhanced J2EE Support for Using WebLogic JMS With EJBs and Servlets

Usability features that are generally hidden behind the J2EE standard have been enhanced to make it easier to access EJB and servlet containers with WebLogic JMS or third-party JMS providers. In fact, implementing this "JMS wrapper" support, as described in this section, is the best practice method of sending a WebLogic JMS message from inside an EJB or servlet.

"Simplified Access to Remote or Foreign JMS Providers" on page 8-15 briefly describes the Administration Console support for foreign JMS providers. 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.

Enabling WebLogic JMS Wrappers

WebLogic Server uses JMS wrappers that make it easier to use WebLogic JMS inside a J2EE component, such as an EJB or a servlet, while also providing 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.

"What's Happening Under the JMS Wrapper Covers" on page 8-5 describes how WebLogic Server implements these features behind the scenes.

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

You enable these enhanced J2EE features by declaring a JMS connection factory as a `resource-ref` in the EJB or servlet deployment descriptors, as described in "Declaring a Wrapped JMS Connection Factory" on page 8-2. For example, when a connection factory is declared as a `resource-ref`, a JMS application can look it up from JNDI using the `java:comp/env/` subtree that is created for each EJB or servlet. It is important to note that the features listed above are only enabled when using a JMS resource inside the deployment descriptors. The EJB and servlet programmers still have direct access to the JMS provider by performing a direct JNDI lookup of the connection factory or destination.

For more information about packaging EJBs, see "Implementing Enterprise JavaBeans" in *Programming WebLogic Enterprise JavaBeans*. For more information about programming servlets, see "Creating and Configuring Servlets" in *Programming WebLogic HTTP Servlets*.

Declaring a Wrapped JMS Connection Factory

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" on page 8-8.

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 will be bound into JNDI, at the location:

```
java:comp/env/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 `weblogic-ejb-jar.xml` (for EJBs) or `weblogic.xml` (for servlets) file that tells the J2EE 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, simply 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

You can also bind a JMS queue or topic destination into the `java:comp/env/jms` JNDI tree by declaring it as a `resource-env-ref` element in the `ejb-jar.xml` or `web.xml` deployment descriptors. 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 `resource-env-ref` description, without having to recompile the source code.

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/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 which allows you to easily modify referenced destinations just by changing the corresponding `resource-env-ref` deployment descriptors.

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

The queue or topic destination specified here must already exist in the global JNDI tree. Again, if the destination does not exist, the application is deployed, but an exception is thrown when you try to use the destination.

Sending a JMS Message In a J2EE Container

After you declare the JMS connection factory and destination resources in the deployment descriptors so that they are mapped to the `java:comp/env` JNDI tree, you can use them to send and/or receive JMS messages inside an EJB or a servlet.

For example, the following code fragment sends a message:

```
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 J2EE specification and should run on any EJB or servlet product that properly supports J2EE — 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" on page 8-8.

Note that this code fragment 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 since WebLogic Server pools some of the objects that are created by this code fragment, 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 fragment. Instead, the container uses them internally if the JMS code is used inside a transaction context. But whether 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 J2EE. 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.

Caution: 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, then the EJB must not use the transactional XA interfaces.

What's Happening Under the JMS Wrapper Covers

This section explains 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 J2EE Container" on page 8-4, 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 J2EE behavior, as described in the following sections.

Automatically Enlisting Transactions

This feature 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, 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, 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 via a `resource-ref` element in the `weblogic-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 J2EE 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 (see "Simplified Access to Remote or Foreign JMS Providers" on page 8-15). Otherwise, WebLogic Server connects to the provider with no user name or password specified. In this mode, it is an error to pass a user name and password to the `createConnection()` method of the JMS 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()` method of the JMS connection factory, or use the version of `createConnection()` with no user name or password if none are required.

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 two minutes by sending a message to a temporary queue or topic and then receiving it again.

J2EE Compliance

The J2EE specification states that you should not be allowed to make certain JMS API calls inside a J2EE 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" on page 8-6; otherwise, it is not. By default, the acknowledgement mode is set to "auto acknowledge". This behavior is expected by the J2EE 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 "Designing and Developing Message-Driven Beans" in *Programming WebLogic Enterprise JavaBeans*.

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 J2EE Container" on page 8-4 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 Connection Factory" on page 8-2.

Improving Performance Through Pooling

The automatic pooling of connections and other objects by the JMS wrappers means that it is efficient to write code as shown in "Sending a JMS Message In a J2EE Container" on page 8-4. Although in this example the Connection Factory, Connection, and Session objects are created every time a message is sent, in reality these three classes work together so that when they are used as shown, they do little more than retrieve a Session object from the pool.

Speeding Up JNDI Lookups by Pooling Session Objects

The JNDI lookups of the Connection Factory and Destination objects can be expensive in terms of performance. This is particularly true if the Destination object points to a Foreign JMS Destination MBean, and therefore, is a lookup on a non-local JNDI provider. Because the Connection Factory and Destination objects are thread-safe, they can be looked up once 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 may 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. Since stateless session beans are pooled, this method is also very efficient (and is perfectly consistent with the J2EE 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 J2EE 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 there has been 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* will have to perform the JNDI lookup of the objects from *Server B* again after that server has recovered. The example, "PoolTestBean.java" on page 8-13 includes a sample EJB that performs this caching and relookup process correctly.

Speeding Up Object Creation Through Caching

Once Connection Factory object and/or Destination object pooling has been 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, 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, 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.

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

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

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
 "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
 "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<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 J2EE container which JMS connection factory and destination to put in that location.

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar PUBLIC
 "-//BEA Systems, Inc.//DTD WebLogic 8.1.0 EJB//EN"
 "http://www.bea.com/servers/wls810/dtd/weblogic-ejb-jar.dtd">

<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>
```

```
  <reference-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>
  </reference-descriptor>
  <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
   // may become invalid if the destination server has
   // been shut down
   qcf = null;
   destination = null;
   throw new RemoteException("Failure in EJB: " + e);
  }
  return id;
 }
}
```


Simplified Access to Remote or Foreign JMS Providers

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 *WebLogic Server MBean Reference*.

- 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 *WebLogic Server MBean Reference*.
- 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. For more information see, "ForeignDestinationBean" in the *WebLogic Server MBean Reference*.

For information on how to configure foreign resources using the Administration Console, see Configuring JMS System Resources in *Configuring and Managing WebLogic JMS*.

Once deployed, these *foreign* System Module MBeans work by creating objects in the local server's JNDI tree, which then perform the lookup of the referenced remote JMS objects whenever the foreign System Module MBeans are looked up. This means that the local server and the remote JNDI directory are never out of sync. However, from a performance perspective, it means that a JNDI lookup of one of these MBeans can potentially be expensive. The sections under "Improving Performance Through Pooling" on page 8-8 describes some ways to improve the performance of these remote lookups.

CHAPTER 9o

Using Message Unit-of-Order

The following sections describe 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 value-added 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

The following sections compare message processing as described by the JMS specification with message processing enhanced by using WebLogic Server's Message Unit-of-Order feature.

Message Processing According to the JMS Specification

While the Java Message Service Specification provides an ordered message delivery, it does so in a very strict sense. It defines order between a single instance of a producer and a single instance of a consumer, but does not take into account the following common situations:

- Distributed Queues were multiple producers within a single application acting as a single producer or multiple consumers. See "Using Distributed Destinations" on page 7-1.
- 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?" on page 9-13.
- Use of filters and destination sort keys. See "Message Unit-of-Order Advanced Topics" on page 9-12.

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.

Delivery Guarantees

Message Unit-of-Order provides the following message delivery guarantees:

- 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 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 Queues" on page 9-14.
- All 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 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 another can be delivered to any queue consumer.

  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.

  – 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.

  – outside of a transaction and the messages are acknowledged (processing is complete). Then the undelivered message M_{n+1} can be processed by any consumer.

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.

- 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, Web Services applications can use conversation names and GUIDs (provided `hashcode` and `equals` for HashMap keys are observed).
- 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. The Unit-of-Order name can be extracted from a delivered message. Example:

  ```
  weblogic.jms.extensions.WLMessageProducer.setUnitOfOrder(weblogic.jm
  s.Message.getStringProperty("JMS_BEA_UnitOfOrder"))
  ```

  So a system-generated Unit-of-Order Name can be used on 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 has been sent through a non-Unit-of-Order JMS provider (releases prior to WebLogic 9.0 or non-Weblogic JMS providers).

Delivery Guarantees

Message Unit-of-Order provides the following message delivery guarantees:

- 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 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 Queues" on page 9-14.
- All 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 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 another can be delivered to any queue consumer.

  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.

  - 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.
  - outside of a transaction and the messages are acknowledged (processing is complete). Then the undelivered message M_{n+1} can be processed by any consumer.

Acknowledgement Rules

This section provides information on rules for JMS acknowledgement modes when using Message Unit-of-Order:

- When the consumer is closed, the current message processing is completed, regardless of the session's acknowledge mode.
- `CLIENT_ACKNOWLEDGE` – The application calling `WLMessage.acknowledge` and `Session.recover(msg)` 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 `onMessage(msg)` indicates when a message is completely processed.
  - Synchronous mode: For consumer given consumer, such as consumer A, `consumerA.receive` is completed when one of the following occurs: `consumerA.receive`, `consumerA.setListener`, 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)` unambiguously indicates when a message is completely processed.
  - Synchronous mode: For consumer given consumer, such as consumer A, `consumerA.receive()` is completed when one of the following occurs: `consumerA.receive()`, `consumerA.setListener()`, or `consumerA.close()`.
- `NO_ACKNOWLEDGE` – Although the session does not acknowledge the receipt of a message or recovery of a message, a message is completed when a call to `receive` or when the `MessageListener` that was called returns successfully.

  - Asynchronous mode: Successful completion or exception of `onMessage(msg)` unambiguously indicates when a message is completely processed.
  - Synchronous mode: For consumer given consumer, such as consumer A, `consumerA.receive()` is completed when one of the following occurs: `consumerA.receive()`, `consumerA.setListener()`, or `consumerA.close()`.

Message Unit-of-Order Case Study

This section provides a simple case study for Message Unit-of-Order based on ordering a book from an online bookstore.

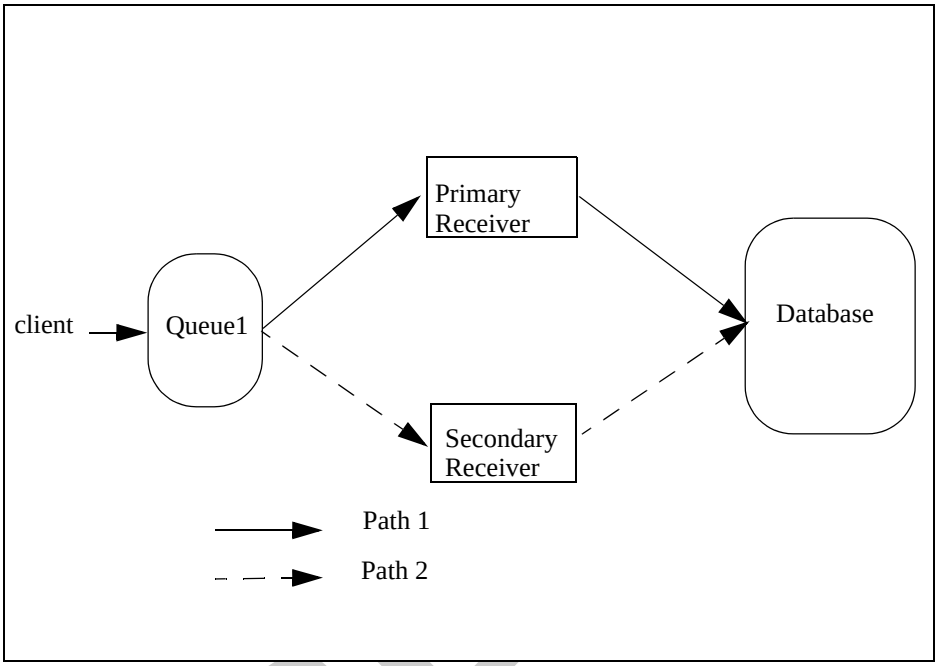
XYZ Online Bookstore Workflow

XYZ Online Bookstore implements a simple processing design that uses web services and JMS to process customer messages. The JMS processing system is composed of a queue (Queue1), a primary and secondary receiver, and a database. XYZ uses a primary receiver to handle order information with a secondary receiver available to provide load balancing and failover capabilities. For this case study, writing to the database is equivalent to successfully processing a message. Messages are processed using the following steps:

1. Customer enters a message using a web services client
2. The web services client places the message on Queue1
3. Message is removed from Queue1 by a receiver.
4. The receiver writes the message to the database.

The message flow of the system is shown below in Figure 9-1.

Figure 9-1 XYZ Online Bookstore Order Processing System



Scenario - Joe Orders a Book

Joe wishes to purchase a book from XYZ Online Bookstore. He logs into his account and searches his favorite book topics. After viewing several selections, Joe decides to purchase one of the books. He proceeds to the checkout and completes the sales transaction. Realizing he has recently purchased this item, Joe cancels the order. One week later, the book is delivered to Joe.

Explanation of What Happened to Joe's Order

In Joe's ordering scenario, his cancel order message was processed before his 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:

1. Joe clicks the order button from his shopping cart.
2. The order message is placed on Queue1

3. Joe cancels the order
4. The cancel order is placed on Queue1
5. The Primary Receiver takes the order message from Queue1
6. The Secondary Receiver takes the cancel order from Queue1 as the Primary Receiver was not available. This may have occurred because the order message takes longer to process before writing to the database.
7. The Secondary Receiver writes the cancel message to the database. Since there is no corresponding order message, there is no order message to remove from the database.
8. The Primary Receiver 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 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, the Primary Receiver was not available to process the cancel order message and the order of the messages was no longer guaranteed.

How Message Unit-of-Order Solves the Problem

Message Unit-of-Order guarantees that Joe's order message is processed before the cancel message is processed. WebLogic can use various means to insure that all messages in Joe's Unit-of-Order are processed sequentially in the order they were created, such as:

- Blocking the delivery of the cancel message until the order message has been processed.
- Ensuring that Joe's messages use the same receiver during processing (have the same producer and consumer).

The following steps demonstrate how Joe's order is processed using Message Unit-of-Order:

1. Joe logs in to his account and a Message Unit-of-Order is created programmatically based on his profile.
2. Joe clicks the order button from his shopping cart.
3. The order message is placed on Queue1.
4. Joe cancels the order.
5. The cancel order is placed on Queue1.

6. The Primary Receiver takes the order message from Queue1.
7. The Secondary Receiver takes the cancel order from Queue1.
8. If the order message is not yet processed, the cancel message on the Secondary Receiver is blocked until processing is complete.
9. The order message is written to the database.
10. The cancel message is written to the database. Since there is a corresponding order message, Joe's order message is removed from the database.
11. Joe's order is cancelled and he does not receive a book.

How to Create a Unit-of-Order

The following sections describe how to create a Message Unit-of-Order. Also see "Unit-of-Order Naming Rules" on page 9-3 and "Message Unit-of-Order Advanced Topics" on page 9-12.

- "Creating a Unit-of-Order Programmatically" on page 9-9
- "Creating a Unit-of-Order Administratively" on page 9-12

Creating a Unit-of-Order Programmatically

There is a WebLogic API extension to the JMS MessageProducer interface that you use to associate a producer with a Unit-of-Order name. Once 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 following code provides an example of how to associate a producer with a Unit-of-Order:

Listing 9-1 Using the MessageProducer Interface to Create a Unit-of-Order

```
.
.
.
/*******************************************
   * createmyproducer
   *******************************************/
```

```
  public void createmyproducer(MessageProducer mp, String name, String uoo,
String pfx)
          throws JMSException {
    this.producer = (WLMessageProducer)mp;
    this.name     = name;
    this.txtmsgprefix = pfx + " ";
    this.ll_myUOOMessages  = new LinkedList();
    this.ll_myUOOMessages.clear();

    setUOO(uoo);
    String gottenUooname = getUOO();
    this.uooname  = gottenUooname != null ? gottenUooname : "<null>";
    inform("Created UOOProducer : ");
    inform("    Name  : " + name);
    inform("    UOO   : " + this.uooname);
    inform("    Prefix: " + pfx);
    ll_allUOOProducers.add(this);
    }    // createmyproducer


  /*****************************************
   * setUOO - set the UOO name of this producer
   *****************************************/
  public boolean setUOO(String uoo) throws JMSException {
    if ( isNonUOOmode )
      getProducer().setUnitOfOrder(null);
    else if ( uoo != null && uoo.equals(SYSTEM) )
      getProducer().setUnitOfOrder();
```

```
    else if ( uoo != null && uoo.equals(DO_NOT_SET_UOO) ) {
      inform("*** Not Setting UOO for producer " + this.name + "
explicitely!");
      return true;
      }
    else {
     inform("Setting UOO for producer " + this.name + " to \"" + uoo + "\"");
      getProducer().setUnitOfOrder(uoo);
      }

    String gottenUooname = getProducer().getUnitOfOrder();
    this.uooname  = gottenUooname != null ? gottenUooname : "<null>";
    return true;
    }  // setUOO
.
.
.
/*****************************************
   * getUOO - get the UOO name of this producer
   *****************************************/
  public String getUOO() throws JMSException {
    return  getProducer().getUnitOfOrder();
    }    // getUOO ends


  /*****************************************
   * return this.producer
   *****************************************/
```

```
  public WLMessageProducer getProducer() {
    if ( this.producer == null ) createConnectionSessionProducers();
    return this.producer;
    }  // getProducer()
.
.
.
```

Creating a Unit-of-Order Administratively

Use a JMS connection factory to associate a Unit-of-Order with all sessions created from factory configured with Unit-of-Order enabled, and, optionally, a name provided. As a result, all sessions created from this connection factory will have Unit-of-Order enabled. For detailed information on creating a Message Unit-of-Order using the administration console, see [link to console help goes here].

All messages produced from the same session will belong to the same Unit-of-Order. Messages from different sessions belong to different Units of Order. Of course, a client can call `WLProducer.setUnitOfOrder(name)` or `setUnitOfOrder` to change the initial connection factory setting on the producer.

You should use this method to create a Unit-of-Order when you need to:

- Enable the Message Unit-of-Order feature on legacy JMS applications without making code changes.

Message Unit-of-Order Advanced Topics

The following sections describe how Unit-of-Order processes messages in advanced or more complex situations:

- "What Happens When a Message Is Delayed During Processing?" on page 9-13
- "What Happens When a Filter Makes a Message Undeliverable" on page 9-13
- "What Happens When Destination Sort Keys are Used" on page 9-13
- "Using Unit-of-Order with Distributed Queues" on page 9-14

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 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 reason listed above. Even though nothing is delaying the delivery of message B, it is not deliverable until message A in its Unit-of-Order has been 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. However, message D is presented to Consumer1 and it does not pass the filter. Messages E through Z are undeliverable until Consumer1 is finished processing message D. See "Filtering Messages" on page 5-29.

What Happens When Destination Sort Keys are Used

Destination sort keys control the order in which messages are presented to consumers when messages are not part of a Unit-of-Order or are not part of the same Unit-of-Order.

For example:

> Messages A and B arrive in that order in the same Unit-of-Order on a queue that is sorted by priority, 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 has been 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, the priority setting of message C and the priority setting of message A determine the delivery order. See Configuring JMS System Resources in *Configuring and Managing WebLogic JMS*.

Using Unit-of-Order with Distributed Queues

As previously discussed in the "Message Processing According to the JMS Specification" on page 9-2, the Java Message Service Specification does not guarantee ordered message delivery when applications use distributed queues. WebLogic Message Unit-of-Order ensures that messages sent to a distributed queue are processed sequentially in the order the messages were created by sending all messages for a Unit-of-Order to the same distributed queue member. If Message Unit-of-Order is configured, when the server processes the first message of a Unit-of-Order that has a distributed queue as a destination, it selects a distributed queue member that will process all remaining messages in that Unit-of-Order.

The following sections provide information on how WebLogic determines the routing path to a distributed queue member:

Persistent Path Routing

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 distributed queue. If the WebLogic Path Service is configured for a distributed queue, the routing path to a member queue is determined by the server using the run-time load balancing heuristics for the distributed queue. See Using WebLogic Path Service in *Configuring and Managing WebLogic JMS*.

The first time a Unit-of-Order is routed to a distributed queue, the server creates a path entry containing the name of the distributed queue, the Unit-of-Order name, and the physical queue member. If one or more producers send messages using the same Unit-of-Order name, all messages they produce will share the same path entry and have the same member queue destination. If the required route for a Unit-of-Order name is unreachable, the producer sending the message will throw a `JMSOrderException`. A path entry can be deleted when the last producer and last message reference are deleted. heuristics

You can use the `weblogic.jms.extensions.WLSession.deleteUnitOfOrderPath` to delete a Unit-of-Order on a distributed queue. During message processing, an application can acquire knowledge that the Unit-of-Order name, such as a workflow or conversation, will no longer produce messages. This provides a mechanism for a distributed queue to easily remove a Unit-of-Order path entry. After the path entry is deleted, the same Unit-of-Order name can be sent to a different distributed queue member.

Implementing Message Unit-of-Order with Persistent Routing

Consider the following when implementing Message Unit-of-Order in conjunction with Path Service-based routing:

- Depending on your system, using the Path Service may slow system throughput due to additional disk operations to create, read, and delete path entries.
- A distributed queue and its individual members each represent a unique destination. For example:

  DXQ1 is a distributed queue with queue members Q1 and Q2. DXQ1 also has a Unit-of-Order name value of *Fred* mapped by the Path Service to queue Q2.

  – DXQ1, Q1, and Q2 are three unique destinations.

  – If message M1 is sent directly to queue Q2, no routing by the Path Service is performed. This is because the route is direct to the member and does not require the system to pin a member from a distributed destination.

  – If message M1 is sent to DXQ1, even though it is ultimately consumed by Q2, it uses the Path Service to define a route and it is not the same route used to send a message directly to directly to queue Q2.

  – You can have more than one destination that has the same Unit-of-Orders name in a distributed queue. For example:

    Queue Q3 has a Unit-of-Order name value of *Fred*. If Q3 is added to DXQ1, there are two destinations that have the same Unit-of-Order name in a distributed queue. Even though, queue Q3 and distributed queue DXQ1 share the same Unit-of-Order name value *Fred*, each has a unique route and destination that allows the server to continue to provide the correct message ordering for each destination.

- Empty queues before removing them from a distributed queue. Although the Path Service will remove the path entry for the removed member, there is a short transition period where a message produced may throw a `JMSOrderException` when the queue has been removed but the path entry still exists.

Non-Persistent Path Routing

If the WebLogic Path Service is not configured, the routing path to a member queue is chosen by the server based on the hash codes of the Message Unit-of-Order name and the distributed queue members. If the required route for a Unit-of-Order name is unreachable, the producer sending the message will throw a `JMSOrderException`. 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. However, 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.

Limitations to Message Unit-of-Order

This section provides additional general information to consider 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-0f-Order does not support clients connecting to a non-Unit-of-Order JMS provider (releases prior to WebLogic 9.0 or non-WebLogic JMS providers).
- Set the `PreserveMsgProperty` on the Messaging Bridge to preserve the Unit-of Order name when connecting between WebLogic 9.0 server instances using the WebLogic Message Bridge. See *Configuring and Managing WebLogic Messaging Bridge*.

CHAPTER 10

Using Transactions with WebLogic JMS

The following sections describe how to use transactions with WebLogic JMS:

Note: For more information about the JMS classes described in this section, access the latest JMS Specification and Javadoc supplied on the Sun Microsystems' Java Web site at the following location: http://java.sun.com/products/jms/docs.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 actually delivered. If the application rolls back the transaction, 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 may 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.

To enable multiple JMS servers in the same JTA user transaction, or to combine JMS operations with non-JMS operations (such as EJB), the two-phase commit license is required. For more information, see "Using JTA User Transactions" on page 10-4.

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 Session Exception Listener" on page 5-14.

If the `acknowledge()` method is called within a transaction, it is ignored. If the `recover()` method is called within a transaction, a JMSException is thrown.

Using JMS Transacted Sessions

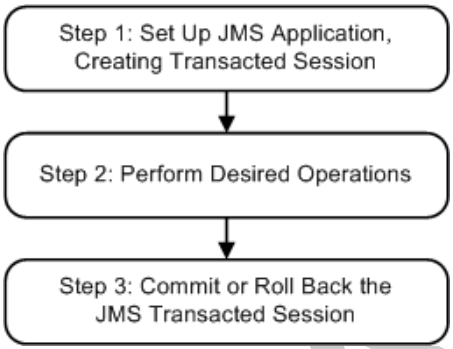
A JMS transacted session supports transactions that are located within the session. A JMS transacted session's transaction will not have any effects outside of the session. For example, rolling back a session will roll back all sends and receives on that session, but will not roll back any database updates. 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.

The following figure illustrates the steps required to set up and use a JMS transacted session.

Figure 10-1 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" on page 4-3, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-7, specify that the session is to be transacted by setting the `transacted` boolean value to `true`.

For example, the following methods illustrate how to create a transacted session for the PTP and Pub/sub messaging models, respectively:

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

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

Once 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 assoicated with the current transaction.

Step 3: Commit or Roll Back the JMS Transacted Session

Once 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, 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 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 *Programming WebLogic JTA*.

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

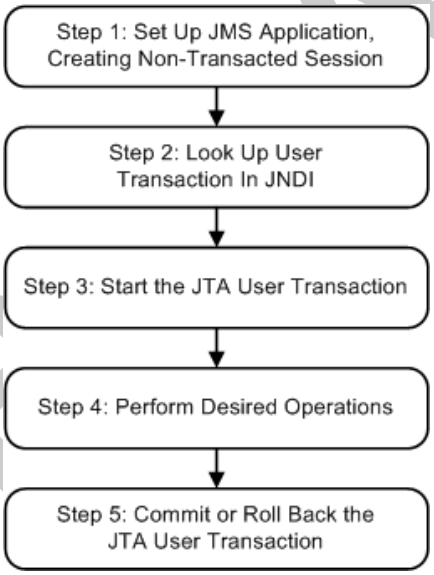
integrity by ensuring that transactional updates are committed in all of the participating resource managers, or are fully rolled back out of all the resource managers, reverting to the state prior to the start of the transaction.

Note: A separate 2PC transaction license is required to support this protocol. For transaction migration considerations related to 2PC, see "Porting WebLogic JMS Applications" on page 13-1.

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.

The following figure illustrates the steps required to set up and use a JTA user transaction.

Figure 10-2 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" on page 4-3, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-7, 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 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()`.

Asynchronous Messaging Within JTA User Transactions Using Message Driven Beans

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 prior to message delivery.

For information on using message driven beans to simulate asynchronous message delivery, see "Designing Message-Driven Beans" in *Programming WebLogic EJB*.

Example: JMS and EJB in a JTA User Transaction

The following example shows how to set up an application for mixed EJB and JMS operations in a JTA user transaction 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, refer to the example provided with WebLogic JTA, located in the *WL_HOME*`\samples\server\examples\src\examples\jta\jmsjdbc` directory, where where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

Import the appropriate packages, including the `javax.transaction.UserTransaction` package.

```
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, creating a non-transacted session. For more information on setting up the JMS application, refer to "Setting Up a JMS Application" on page 4-3.

```
//JMS application setup steps including, for example:

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

Step 2

Look up the `UserTransaction` using JNDI.

```
UserTransaction xact = (UserTransaction)
        ctx.lookup(JTA_USER_XACT);
```

Step 3

Start the JTA user transaction.

```
xact.begin();
```

Step 4

Perform the desired operations.

```
// Perform some JMS and EJB operations here.
```

Step 5

Commit the JTA user transaction.

```
xact.commit()
```

CHAPTER 11

WebLogic JMS C API

The following sections describe how to deploy and use the WebLogic JMS C API:

What is the C API

The Weblogic JMS C API enables programs written in C to participate in JMS applications using the Java Native Interface (JNI) to access a Java Virtual Machine (JVM). For this release, the JMS C API adheres to the JMS Version 1.1 specification to promote the porting of Java JMS 1.1 code. For more information, see the WebLogic JMS C API Javadocs.

Operating System Requirements

See the List of Supported Operating System Configurations to view operating systems supported by BEA for this release of the WebLogic JMS C API.

Configuring WebLogic Server to use the C API

To properly configure your environment to use the C API, do the following:

- Set your system's dynamic library path to include the Java Runtime Environment's JNI library and the JMS C API library.
- Set the CLASSPATH environment variable as if you were running a Java JMS client.
- Copy the `JmsTypes.h` file for your platform into the *`WL_HOME`*`\server\include` directory.
  - Windows—copy the file located at `WL_HOME\server\include\platform\win32`
  - Solaris—copy the file located at `WL_HOME/server/include/platform/sol`
  - HP-UX—copy the file located at `WL_HOME/server/include/platform/hp-ux`
  - Redhat Linux AS 2.1 32-bit Pentium—copy the file located at `WL_HOME/server/include/platform/lnxi686`
  - Redhat Linux AS 2.1 64-bit Itanium—copy the file located at `WL_HOME/server/include/platform/lnxia64`

BEA Systems provides samples for JMS developers which illustrate how to configure and develop the WebLogic JMS C API clients. The JMS C API samples are available for download at http://dev2dev.bea.com/code/certwls90.jsp.

Design Principles

The idea is that if a resource was allocated to the programmer, the programmer must also dispose of it properly.

The following sections discuss guiding principals developers should consider when porting or developing applications for the WebLogic JMS C API:

- "Java Objects Map to Handles" on page 11-3
- "Thread Utilization" on page 11-3
- "Exception Handling" on page 11-3
- "Type Conversions" on page 11-4
- "Garbage Collection" on page 11-5
- "Closing Connections" on page 11-6

- "Helper Functions" on page 11-6

Java Objects Map to Handles

The JMS C API is handle-based to promote modular code implementation. This means that in your application, Java objects are implemented as handles in C code. The details of how a JMS object is implemented is hidden inside a handle. However, unlike in Java, all handles given to the user of the API must be explicitly freed by calling the corresponding `Close` or `Destroy`. See "Garbage Collection" on page 11-5.

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 application, all objects returned by the WebLogic JMS C API may 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`—which represents the class `java.lang.Throwable`
- `JavaException`—which represents the class `java.lang.Exception`
- `JmsException`—which represents the class `javax.jms.JMSException`. All of the 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

Typically, when interoperating between Java code to C code, one of the main tasks is converting a C type to a Java type. For example, a short is a two-byte entity in Java as well as in C. This section provides information on the type conversions that require special handling:

Integer (int)

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

Long (long)

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

Character (char)

`Character (char)` converts to `short` (two byte java character).

String

`String` converts to `JmsString`.

In Java strings are arrays of two byte characters. In C, strings are generally arrays of one-byte UTF-8 encoded characters. Pure ASCII strings fit into the UTF-8 specification as well. For more information on UTF-8 string, see `www.unicode.org`. It is inconvenient for C programmers to translate all strings into the two-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 string:

- native C `string (CSTRING)` or
- `JavaString (UNISTRING)`

A union 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 used as input, the stringType element of JmsString is set 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 two bytes as a single Java character. The two-byte sequence is big-endian. Unicode calls this encoding UTF-16BE (as opposed to UTF-16LE, which is a two-byte sequence that is little-endian). The CSTRING encoding expects a UTF-8 encoded string.

When used as output, the caller has the option to let the API allocate enough space for output using malloc, or to 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:

> To allocate one hundred bytes for the string output from a text message, you would set the data pointer and the allocatedSize field to one hundred. The JmsMessageGetTextMessage API returns JMS_NEED_SPACE with allocatedSize set to two hundred. Call realloc on the original string to reset the data pointer and call the function again. Now the call succeeds and and you are able to extract the string from the message handle. Another method would be to free the the original buffer and allocate a new one of the correct size.

Garbage Collection

In Java, garbage collection cleans up all objects that are no longer referenced. However, in C, all objects must be explicitly cleaned up. This requires 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 is used to distinguish between Java objects that have a close method and those that do not. 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 WebLogic JMS C API. 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 hashtable used to create the InitialContext object.

Limitations and Guidelines

The following section provides guidelines and limitations that you should consider when implementing 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.

- If the user wishes to translate the message catalog they can use the gencat utility provided on Windows or the gencat utility of the host platform. If the generate 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.
- The following environment variables can be set in the client environment:
  - JMSDEBUG— Use to provide verbose debugging output from the client.
  - JMSJVMOPTS—Use to provide extra arguments to the JVM loaded by the client.
  - ULOGPFX—Use to set to the pathname and file prefix where the error log file is placed.

Recovering from a WebLogic Server Failure

The following sections describe how to terminate a JMS application gracefully if a server fails and how to migrate JMS data after server failure.

Programming Considerations

You may want to program your JMS application to terminate gracefully in the event of a WebLogic Server failure. For example:

Table 12-1 Programming Considerations for Server Failures

If a WebLogic Server Instance Fails and...	Then...
You are connected to the failed WebLogic Server instance	A `JMSException` is delivered to the connection exception listener. You must restart the application once the server is restarted or replaced.
A JMS Server is targeted on the failed WebLogic Server instance	A `ConsumerClosedException` is delivered to the session exception listener. You must re-establish any message consumers that have been lost.

Migrating JMS Data to a New Server

WebLogic JMS uses the migration framework implemented in the WebLogic Server core, which allows WebLogic JMS 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.

Once properly configured, a JMS server and all of its destination members can migrate to another WebLogic Server within a cluster.

You can recover JMS data from a failed WebLogic Server by starting a new server and doing one or more of the tasks in Table 12-2.

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 When Currently Active Host is Unavailable" in *Using WebLogic Server Clusters*.

Table 12-2 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, 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 impacted, and no changes are required.

If Your JMS Application Uses. . .	Perform the Following Task. . .
Persistent messaging—File Store	Migrate the file to the new server, ensuring that the pathname within the WebLogic Server home directory is the same as it was on the original server.
Transactions	Migrate the transaction log to the new server by copying all files named `<servername>*.tlog`. This can be accomplished by storing the transaction log files on a dual-ported disk that can be mounted on either machine, or by manually copying the files. If the files are located in a different directory on the new server, update that server's `TransactionLogFilePrefix` server configuration attribute before starting the new server. **Note:** If migrating following a system crash, it is very important that the transaction log files be available when the server is restarted at its new location. Otherwise, transactions in the process of being committed at the time of the crash might not be resolved correctly, resulting in data inconsistencies. All uncommitted transactions are rolled back.

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, 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.

For information about starting a new WebLogic Server, see the "Starting and Stopping Servers: Quick Reference". For information about recovering a failed server, refer to WebLogic Server Availability and Failure Recovery Features in *Managing Server Startup and Shutdown*.

For more information about defining migratable services, see "Migration for Singleton Services" in *Using WebLogic Server Clusters*.

CHAPTER 13

Porting WebLogic JMS Applications

The following sections describe how to port your WebLogic JMS applications to a newer version of WebLogic Server:

Existing Feature Functionality Changes

Changes in existing feature functionality have been made in order to comply with Sun Microsystems' JMS Specification. Therefore, you should check feature functionality changes in the following tables before beginning any porting procedures:

Existing Feature 5.1 to 6.0 Functionality Changes

The following table lists the changes in existing feature functionality from WebLogic Server version 5.1, and also indicates any code changes that might be required as a result. For additional

information pertaining to the JMS Specification's version change history, refer to Chapter 11, "Change History" in the specification.

Category	Description	Code Modification

Connection Factories	Two default connection factories have been deprecated. The JNDI names for these factories are:
	• `javax.jms.QueueConnectionFactory`
	• `javax.jms.TopicConnectionFactory`
	For backwards compatibility, the JNDI names for these two connection factories are still defined and supported.
	
	WebLogic JMS 6.*x* or later defines one connection factory, by default: `weblogic.jms.ConnectionFactory`
	You can also specify user-defined connection factories using the *Administration Console*.
	Note: Using the default connection factories, you have no control over the WebLogic server on which the connection factory may be deployed. However, you can can enable and/or disable the default connection factories on a per WebLogic Server basis. To deploy a connection factory on a particular WebLogic Server or cluster, create a new connection factory and specify the appropriate WebLogic Server target.
		It is recommended that existing code that use the deprecated classes be modified to use a new default or user-defined connection factory class.
		For example, if your code specified the following constant using the default queue connection factory:
		`public final static String JMS_FACTORY="javax.jms.QueueConnectionFactory"`
		You should modify the constant to use a new user-defined connection factory, for example:
		`public final static String JMS_FACTORY="weblogic.jms.QueueConnectionFactory"`
		For true backwards compatibility with previous releases, you should ensure that you select the Allow Close In onMessage and User Transactions Enabled check boxes when configuring the connection factory.
		For more information about using connection factories, see "ConnectionFactory Object" on page 2-9.

	In order to instantiate the default connection factories on a particular WebLogic Server, you must select the Enable Default JMS Connection Factories check box on the Server --> Services --> JMS node tab when configuring the WebLogic Server.	None required. This is a configuration requirement.
Connections	When closing a connection, the call blocks until outstanding synchronous calls and asynchronous listeners have completed.	None required.

Category	Description	Code Modification
Sessions	When closing a session, the call blocks until outstanding synchronous calls and asynchronous listeners have completed.	None required.
Message Consumers	If multiple topic subscribers are defined in the same session for the same topic, each consumer will receive its own copy of a message.	None required.
	When closing a message consumer, if the consumer is asynchronous then the call blocks until outstanding calls to the `onMessage()` method complete.	None required.
	In order to comply with the JMS specification, if the `close()` method is called from within an `onMessage()` method, the application will hang unless the Allow Close In OnMessage check box is selected when configuring the connection factory. If the acknowledge mode is `AUTO_ACKNOWLEDGE`, the current message will still be automatically acknowledged.	None required. This is a configuration requirement.

Category	Description	Code Modification

Message Header Field	The `JMSMessageID` header field format has changed.	If you wish to access existing messages using the `JMSMessageID`, you may need to run one of the following `weblogic.jms.extensions.JMSHelper` methods to convert between WebLogic pre-JMS 5.1 and JMS 6.*x* JMSMessageID formats.
		To convert from pre-5.1 to 6.*x* `JMSMessageID` format:
		`public void oldJMSMessageIDToNew( String id, long timeStamp ) throws JMSException`
		To convert from 6.1 to pre- 6.1 `JMSMessageID` format:
		`public void newJMSMessageIDToOld( String id, long timeStamp ) throws JMSException`

Category	Description	Code Modification
Destinations	The `createQueue()` and `createTopic()` methods do not create destinations dynamically, only references to destinations that already exist given the vendor-specific destination name.	Update any portion of code that uses `createQueue()` or `createTopic()` to dynamically create destinations using the following JMSHelper class methods, respectively: `createPermanentQueueAsync()` and `createPermanentTopicAsync()`. For example, if your code used the following method to dynamically create a queue: `queue=qsession.createQueue(queueName);` You should modify the code to dynamically create a queue, as described in the sample `findQueue()` method in "Using the JMS Module Helper Methods to Manage Your Applications" on page 5-34. For more information on the JMSHelper classes, see "Dynamically Creating Destinations" on page 5-34.
	When creating temporary destinations, you must specify a temporary template.	None required. This is a configuration requirement.
	You must be the owner of the connection in order to create a message consumer for that temporary destination.	When creating a message consumer on a temporary destination, ensure that you are the owner of the connection.
Durable Subscribers	You no longer need to manually create JDBC tables for durable subscribers. They are created automatically.	None required.
	There is no limit on the number of durable subscribers that can be created.	None required.
	When defining a client ID programatically, it must be defined *immediately* after creating a connection. Otherwise, an exception will be thrown and you will be unable to make any other JMS calls on that connection.	Ensure that the `setClientID()` method is issued immediately after creating the connection. For more information, refer to "Defining the Client ID" on page 5-18.

Category	Description	Code Modification
Session Pools	Session pool factories, session pools, referenced connection factories, referenced destinations, and associated connection consumers must all be targeted on the same JMS server.	None required. This is a configuration requirement. Ensure that all objects are targeted on the same JMS server.
	The `SessionPoolManager` and `ConnectionConsumerManager` interfaces that were published as part of the WebLogic JMS version 5.1 Javadoc have been removed from the version 6.*x* and later Javadoc, as they are system interfaces and should not be used within client applications.	If used, remove any references to these objects from the client application.
Transactions	To combine JMS and EJB database calls within the same transaction, a two-phase commit (2PC) license is required. In previous releases of WebLogic Server, it was possible to combine them by using the same database connection pool.	None required.
	Recovering or rolling back received queue messages makes them available to all consumers on the queue. In previous releases of WebLogic Server, rolled back messages were only available to the session that rolled back the message, until that session was closed.	None required.

Existing Feature 6.0 to 6.1 Functionality Changes

The following table lists the changes in existing feature functionality from WebLogic Server 6.0, and also indicates any code changes that might be required as a result.

For additional information pertaining to the JMS Specification's change history, see Chapter 11, "Change History," of Sun Microsystems' JMS Specification.

Category	Description	Code Modification
Connection Factories	For the Acknowledge Policy attribute in the Administration Console, the new default value of `All` is a work-around to accommodate a change in the JMS Specification. This new default setting represents a change from prior versions of JMS, which internally defaulted to `Previous`, and which did not appear as an option in the Administration Console. As the message acknowledge policy for the connection factory, the Acknowledge Policy attribute only applies to applications that use the `CLIENT_ACKNOWLEDGE` mode for a non-transacted session. • `All` — acknowledge all messages ever received by a given session, regardless of which message calls the acknowledge method. • `Previous` — acknowledge all messages received by a given session, but only up to and including the message that calls the acknowledge method. For more information on message acknowledge modes, refer to "Non-Transacted Session" on page 2-13. **Note:** For connection factories used by MDBs, always set the Acknowledge Policy field to `Previous`. Although the default MDB connection factory already does this, foreign connection factories may not.	If you want to acknowledge only previously received messages, up to and including the message that calls the acknowledge method, change the default Acknowledge Policy setting from `All` to `Previous` .
Destinations	In WLS 6.0, the JMS documentation correctly specifies values of `default`, `true`, and `false` for the `StoreEnabled` attribute of the `JMSDestinationMBean`, even though the software allowed for mixed case characters. Version 6.1 or later requires all lowercase characters for the `StoreEnabled` settings.	None required. This is a configuration requirement.

Porting Existing Applications

This release of WebLogic Server supports Sun Microsystems' JMS Specification. In order to use your existing JMS applications, you must first confirm your version of WebLogic Server, and then perform the appropriate porting procedures provided in this section.

Before You Begin

Before beginning the porting procedure, you should check the following list to confirm whether porting is support for your version of WebLogic Server JMS, and to find out whether special porting rules apply to that release:

- Weblogic Server 5.1 — Customers running SP07 or SP08 should contact BEA Support before porting existing JDBC stores to version 8.1.
  - In order to port object messages, the object classes need to be in the Weblogic Server 8.1 server `CLASSPATH`.
  - For destinations that are not configured in Weblogic Server 8.1, the ported messages will be dropped and the event will be logged.
- WebLogic Server 6.1 and 7.0 — All applications are supported in version 8.1. However, if you want your applications to take advantage of the new highly available JMS Distributed Destination feature, you will need to configure your existing physical destinations (queues and topics) to be part of a single distributed destination set. For more information, "Using Distributed Destinations" on page 7-1.

Steps for Porting Version 5.1 Applications to Version 8.1

Before you can use an existing WebLogic JMS 5.1 application, you must port the WebLogic Server 5.1 configuration and message data to version 8.1 as follows:

1. Properly shut down the old version of WebLogic Server before beginning the porting process.

   Warning: Abruptly stopping the old version of WebLogic Server while messaging is still in process may cause problems during porting. Processing should be inactive before shutting down the old server and beginning the porting to WebLogic Server 8.1.

2. Upgrade the WebLogic Server environment, as described in *Installing WebLogic Platform*.
3. Ported configuration information using the configuration conversion facility.

   During the configuration porting, the following default queue and topic connection factories are enabled:

   - `javax.jms.QueueConnectionFactory`
   - `javax.jms.TopicConnectionFactory`
   - `weblogic.jms.ConnectionFactory`
   - `weblogic.jms.XAConnection Factory`

   The first two connection factories are deprecated, but they are still defined and usable for backwards compatibility. For information on the new default connection factories, see "ConnectionFactory Object" on page 2-9.

   The JMS administrator will need to review the resulting configuration to ensure that the conversion meets the needs of the application. In this case, all of the JMS attributes will be mapped to a single node, as in version 5.1.

   Note: In versions 6.0 or later, JMS queues are defined during configuration, and no longer saved within database tables. Message data and durable subscriptions are stored either in two JDBC tables or via a directory within the file system.

4. Prepare for automatic porting of existing JDBC database stores.

   a. Make a backup of the existing JDBC database.

   b. Ensure that the ported configuration information (see step 2) contains a JDBC database store with exactly the same attributes as the existing store, and that the new JMS servers that use the store define the same destinations and corresponding destination attributes as the existing JMS servers.

   c. If the new JDBC database store already exists, ensure that it is empty.

      The new JDBC database store will be created during the automatic porting, if required.

   d. Ensure that there is twice the amount of disk space required by the JDBC database store available on the system.

      Both the existing and new database information will exist on disk while the porting is performed, doubling the space requirements. Once porting is complete, you can delete the old JDBC database stores, as described in "Deleting JDBC Database Stores" on page 13-10.

5. Update any existing code, as required, to reflect the feature functionality changes described in "Existing Feature 5.1 to 6.0 Functionality Changes" on page 13-1.
6. Start up the WebLogic Server and the existing JDBC database stores will be upgraded automatically.

   Note: If the automatic porting fails for any reason, the automatic upgrade will be re-attempted the next time the WebLogic Server boots.

Deleting JDBC Database Stores

Once the porting is complete, the old JDBC database tables should be removed using the `utils.Schema` utility, described in detail in Appendix B, "JDBC Database Utility."

During porting, a DDL file is generated and stored in the local working directory. The DDL file is named `drop_<jmsServerName>_oldtables.ddl`, where `<jmsServerName>` specifies the name of the JMS server. To delete the JDBC database stores, you pass the resulting DDL file as an argument to the `utils.Schema` utility.

For example, to delete the old JDBC database store from a JMS server named *MyJMSServer*, run the following command:

```
java utils.Schema jdbc:weblogic:oracle weblogic.jdbc.oci.Driver -s server
-u user1 -p foobar -verbose drop_MyJMSServer_oldtables.ddl
```

For more information on the `utils.Schema` utility, see Appendix B, "JDBC Database Utility."

Steps for Porting Version 6.0 Applications to Version 8.1

Before you can use an existing WebLogic JMS 6.0 application, you must port the WebLogic Server 6.0 configuration and message data to version 8.1 as follows

1. Check the connection factory configuration for version 6.0. You may need to modify programs that call the version 8.1 default connection factory so that they load one of the following connection factories:
   - One of the version 6.0 default connection factories.
   - A custom connection factory.
2. Properly shut down the version 6.0 WebLogic Server before beginning the porting process.

   Warning: Abruptly stopping the old version of WebLogic Server while messaging is still in process may cause problems during porting. Processing should be inactive before shutting down the old server and beginning the porting to WebLogic Server 8.1.

3. Upgrade the WebLogic Server environment, as described in *Installing WebLogic Platform*.
4. Update any existing code, as required, to reflect the feature functionality changes described in "Existing Feature 5.1 to 6.0 Functionality Changes" on page 13-1.

   Warning: Before starting the version 8.1 WebLogic Server, you may want to backup your version 6.0 stores. This is because version 6.0 servers *cannot* use 8.1 stores, and any attempts to do so may cause data corruption.
5. Start the version 8.1 WebLogic Server. This server will continue to use the previous version 6.0 stores.

Steps for Porting Version 6.1 or 7.0 Applications to Version 8.1

All WebLogic JMS 6.1 and 7.0 applications are supported in version 8.1. However, if you want your applications to take advantage of the highly available Distributed Destination feature, you need to configure your existing physical destinations (queues and topics) to be part of a single distributed destination set.

For more information on using JMS distributed destinations, see "Using Distributed Destinations" on page 7-1.

APPENDIX A

Configuration Checklists

The following sections provide monitoring checklists for various WebLogic JMS features:

- Server Clusters
- JTA User Transactions
- JMS Transactions
- Message Delivery
- Asynchronous Message Delivery
- Persistent Messages
- Concurrent Message Processing
- Multicasting
- Durable Subscriptions
- Destination Sort Order
- Temporary Destinations
- Thresholds and Quotas

For more information on setting the configuration attributes, see "Configuring JMS System Resources" in *Configuring and Managing WebLogic JMS*.

Server Clusters

To support server clusters, configure the following:

- Target a connection factory to one or more independent server instances, to all servers in a cluster, or to selected servers in a cluster on the Target and Deploy tab on the Connection Factories node
- Target a JMS server to an independent server instance or to migratable target on the Target and Deploy tab on the JMS Servers node

JTA User Transactions

To support JTA user transactions, configure the following:

- Connection factory JTA user transaction mode by selecting the XA Connection Factory Enabled check box under the Configuration—Transactions tab on the Connection Factories node

JMS Transactions

To support JMS transacted sessions, configure the following:

- Connection factory transaction timeout value by setting the Transaction Timeout attribute under the Configuration—Transactions tab on the Connection Factories node
- Session pool transaction mode by selecting the Transacted check box under the Configuration tab on the Session Pools node

Message Delivery

To define message delivery attributes, configure the following:

- Connection factory priority, time-to-live, time-to-deliver, and delivery mode attributes under the Configuration—General tab on the Connection Factories node
- Destination priority, time-to-live, time-to-deliver, and delivery mode override attributes under the Configuration—Overrides tab on the Destinations node
- Destination redelivery delay, redelivery limit, and error destination attributes under the Configuration—Redelivery tab on the Destinations node

Note: These settings can also be set dynamically by the message producer when sending a message or using the set methods, as described in "Sending Messages" on page 4-21.

The destination configuration attributes take precedence over all other settings.

Asynchronous Message Delivery

To define the maximum number of messages that may exist for an asynchronous session and that have not yet been passed to the message listener, configure the following:

- Message maximum attribute under the Configuration—General tab on the Connection Factories node

Persistent Messages

Note: Topic destinations are persistent if — and only if — they have durable subscriptions. For more information about durable subscriptions, see "Setting Up Durable Subscriptions" on page 5-17.

To support persistent messaging, configure the following:

- Create a JMS file store or JMS JDBC store using the Stores node.
- Select the file or JDBC store on the JMS server by using the Store attribute on the Configuration—General tab on the JMS Servers node.

  Note: No two JMS servers can use the same backing store.
- Establish the default message delivery mode by setting one of the following attributes to `Persistent`:
  - Default Delivery Mode attribute under the Configuration—General tab on the Connection Factories node.
  - Delivery Mode Override attribute under the Configuration—Overrides tab on the Destination node. (This setting can also be set to `No Delivery`. If no JMS template is specified for this destination, then `No-Delivery` means the Delivery Mode will not be overridden. Otherwise, `No-Delivery` means that the value comes from the JMS.

  Note: You can also specify persistent as the delivery mode when sending messages, as described in "Sending Messages" on page 4-21.

Concurrent Message Processing

To support concurrent message processing, configure the following:

- Server session pool attributes under the Configuration tab on the Session Pools node

- Connection consumer attributes under the Configuration tab on the Connection Consumers node

Note: Server session pool factories, used for concurrent message processing, are not configurable. WebLogic JMS defines one ServerSessionPoolFactory object, by default: `weblogic.jms.ServerSessionPoolFactory:<name>`, where <*name*> specifies the name of the JMS server on which the session pool is created. For more information about using server session pool factories, refer to "Defining Server Session Pools" on page C-2.

Multicasting

Note: Multicasting applies to topics only.

To configure multicasting on a topic, configure the following:

- Multicast address, multicast port, and multicast time-to-live (TTL) under the Configuration—Multicast tab on the Destination node
- Maximum number of outstanding messages by setting the Messages Maximum attribute under the Configuration—General tab on the Connection Factories node
- Overrun policy used when the number of outstanding messages reaches the Messages Maximum value by setting the Overrun Policy attribute under the Configuration—General tab on the Connection Factories node

Durable Subscriptions

To support durable topic subscriptions, configure the following:

- Create a JMS file store or JMS JDBC persistent store using the Stores node.
- Target the configured store to your JMS server by selecting it from the Store attribute's drop-down list on the JMS Server →Configuration →General tab.

  Note: No two JMS servers can use the same backing store.

- Set the client ID on either a JMS connection factory or in the topic connection, as follows:
  - One method is to configure a connection factory with the client identifier (client ID). For WebLogic JMS, this means configuring a separate connection factory instance for each client ID. Then, on each connection factory, specify a unique client ID for clients with durable subscriptions using the Client ID attribute on the Connection Factory → Configuration →General tab. Applications then look up their own topic connection factories in JNDI and use them to create connections containing their own client IDs.

– The other “preferred” method is for an application that can set its client ID in the topic connection after the connection is created by calling a connection method, as described in “Setting Up Durable Subscriptions” on page 5-17. By using this alternative approach, you can use the default connection factory (if it is acceptable for your application) and avoid the need to modify the configuration information.

Destination Sort Order

To support destination sort order, configure the following:

- Key attributes under the Configuration tab on Destination Keys node
- Destination Keys under Configuration—General tab on Destinations node

Temporary Destinations

To support temporary destinations (queue or topic), configure the following:

- A JMS template for the JMS server (in the same domain) under the Configuration—General tab on the Templates node
- A JMS template to be used by the JMS server for temporary destinations by setting the Temporary Template attribute for the JMS server under the Configuration—General tab on the JMS Servers node

Thresholds and Quotas

To configure thresholds and quotas, configure the following:

- Message and byte thresholds and quotas (maximum number, and high and low thresholds) under the Configurations—Thresholds tab on the JMS Server node
- Message and byte thresholds and quotas (maximum number, and high and low thresholds) under the Configurations—Thresholds tab on the Destination node
- Maximum number of sessions that can be retrieved from a session pool by setting the Sessions Maximum attribute under the Configurations tab on the Session Pools node
- Maximum number of messages that can be accumulated by a connection consumer by setting the Messages Maximum attribute under the Configuration tab of the Consumers node

JDBC Database Utility

The following sections describe JDBC database stores for WebLogic JMS, and how to use the JDBC database utility to regenerate existing JDBC database stores:

Overview

The JDBC `utils.Schema` utility allows you to regenerate new JDBC database stores by deleting the existing versions. Running this utility is usually not necessary, since JMS automatically creates these stores for you. However, if your existing JDBC database stores somehow become corrupted, you can regenerate them using the `utils.Schema` utility. In addition, if your JDBC driver is not supported by WebLogic JMS (as described in "JMS JDBC Stores Tasks" in the *Administration Console Online Help*), then the tables required by JMS must be created manually.

Caution: Use caution when running the `utils.Schema` command as it will delete all existing database tables and then recreate new ones.

About JMS Tables

The JMS database contains two system tables that are generated automatically and are used internally by JMS, as follows:

- `<prefix>JMSStore`

- `<prefix>JMSState`

The prefix name uniquely identifies JMS tables in the backing store. Specifying unique prefixes allows multiple stores to exist in the same database. The prefix is configured via the Administration Console when configuring the JDBC store. A prefix is prepended to table names when:

- The DBMS requires fully qualified names.
- You must differentiate between JMS tables for two WebLogic servers, enabling multiple tables to be stored on a single DBMS.

The prefix should be specified using the following format, which will result in a valid table name when prepended to the JMS table name:

```
[[catalog.]schema.]prefix
```

Note: No two JMS stores should be allowed to use the same database tables, as this will result in data corruption.

For more information on configuring JDBC database stores for WebLogic JMS, see "JMS JDBC Store Tasks" in the *Administration Console Online Help*.

Regenerating JDBC Database Stores

The `utils.Schema` utility is a Java program that takes command line arguments to specify the following:

- JDBC driver
- Database connection information
- Name of a file containing the SQL Data Definition Language (DDL) commands (terminated by semicolons) that create the database tables

By convention, the DDL file has a `.ddl` extension. DDL files are provided for Pointbase, Cloudscape, Informix, Sybase, Oracle, MS SQL Server, IBM DB2, and Times Ten databases.

To execute `utils.Schema`, your `CLASSPATH` must contain the `weblogic.jar` file.

Enter the `utils.Schema` command, as follows:

```
java utils.Schema url JDBC_driver [options] DDL_file
```

The following table lists the `utils.Schema` command-line arguments.

Argument	Description
url	Database connection URL. This value must be a colon-separated URL as defined by the JDBC specification.
JDBC_driver	Full package name of the JDBC Driver class.
options	Optional command options. If required by the database, you can specify: • The username and password as follows: `-u <username> -p <password>` • The Domain Name Server (DNS) name of the JDBC database server as follows: `-s <dbserver>` You can also specify the `-verbose` option, which causes `utils.Schema` to echo SQL commands as they are executed.
DDL_file	The full pathname of a text file containing the SQL commands that you wish to execute. An SQL command can span several lines and is terminated with a semicolon (;). Lines beginning with pound signs (#) are comments. The `weblogic/jms/ddl` directory within the `weblogic.jar` file contains JMS DDL files for Pointbase, Cloudscape, Informix, Sybase, Oracle, MS SQL Server, IBM DB2, and Times Ten databases. To use a different database, copy and edit any one of these files. Use the jar utility supplied with the JDK to extract them to the `weblogic/jms/ddl` directory using the following command: `jar xf weblogic.jar weblogic/jms/ddl` **Note:** If you omit the `weblogic/jms/ddl` parameter, the entire jar file is extracted.

For example, the following command recreates the JMS tables in an Oracle server named `DEMO`, with the username `user1` and password `foobar`:

```
java utils.Schema jdbc:weblogic:oracle:DEMO \
 weblogic.jdbc.oci.Driver -u user1 -p foobar -verbose \
 weblogic/jms/ddl/jms_oracle.ddl
```

With the Pointbase demo database that is shipped with WebLogic Server, no username or password is required. However, you must follow this procedure to create the JMS tables in a Pointbase server:

1. Set the WLS samples environment:

   ```
   %SAMPLES_HOME%\server\config\examples\setExamplesEnv.cmd
   ```

2. Change to the `%WL_HOME%\server\lib\`directory, and then extract the `jms_pointbase.ddl` file from the `weblogic.jar` file to the current directory.

3. Execute the following command to create the JMS tables:

```
java utils.Schema jdbc:pointbase:server://localhost/demo
 com.pointbase.jdbc.jdbcUniversalDriver
 -u examples -p examples -verbose jms_pointbase.ddl
```

   The Pointbase JDBC URL specifies the demo database, which is included with the WebLogic JMS samples. For the samples, the JMS tables have already been created in this database.

4. Start the Pointbase server and open the Pointbase console.

   For detailed information on using the Pointbase Server console to monitor and manipulate the JMS tables, see the Pointbase.html file in the *WL_HOME*`\samples\server\src\examples` directory, where *WL_HOME* is the top-level directory of your WebLogic Platform installation.

APPENDIX C

Deprecated WebLogic JMS Features

The following sections describe features that have been deprecated for this release of WebLogic Server :

Defining Server Session Pools

Note: Session pools are now used rarely, as they are not a required part of the J2EE 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 on designing MDBs, see "Designing and Developing Message-Driven Beans" in *Programming WebLogic Enterprise JavaBeans*.

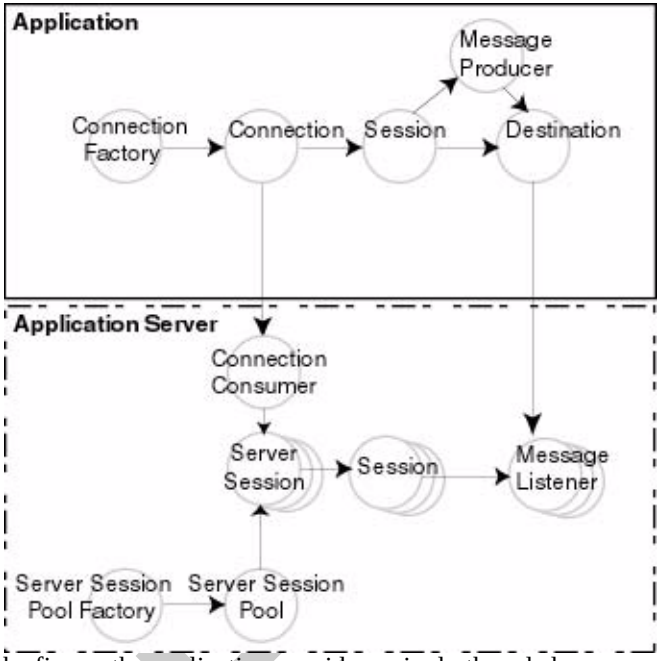
WebLogic JMS implements an optional JMS facility for defining a server-managed pool of server sessions. This facility enables an application to process messages concurrently.

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.

The following figure illustrates the server session pool facility, and the relationship between the application and the application server components.

Figure 0-1 Server Session Pool Facility

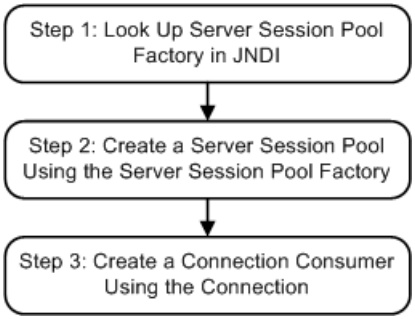


As illustrated in the figure, the application provides a single-threaded message listener. The connection consumer, implemented by JMS on the application server, performs the following tasks to process one or more messages:

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 pool when finished processing messages.

The following figure illustrates the steps required to prepare for concurrent message processing.

Figure 0-2 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 on using message driven beans to implement server session pools, see "Designing Message-Driven Beans" in *Programming WebLogic Enterprise JavaBeans*.

If the session pool and connection consumer were defined during configuration, you can skip this section. For more information on configuring server session pools and connection consumers, see "Configuring JMS System Resources" in *Configuring and Managing WebLogic JMS*.

Currently, WebLogic JMS does *not* support the optional `TopicConnection.createDurableConnectionConsumer()` operation. For more information on this advanced JMS operation, refer to Sun Microsystems' JMS Specification.

Step 1: Look Up 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.

Once it has been configured, you can look up a server session pool factory by first establishing a JNDI context (`context`) using the `NamingManager.InitialContext()` method. 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.

Once 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.

For more information about server session pool factories, see "ServerSessionPoolFactory Object" on page 2-24 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 (PTP) or topic (Pub/Sub) connection consumers, using the `ServerSessionPoolFactory` methods described in the following sections.

For more information about server session pools, see "ServerSessionPool Object" on page 2-25 or the `javax.jms.ServerSessionPool` Javadoc.

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.

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.

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 JMS System Resources" in *Configuring and Managing WebLogic JMS*
- Including in your application the Connection methods described in the following sections

For more information about the `ConnectionConsumer` class, see "ConnectionConsumer Object" on page 2-25 or the `javax.jms.ConnectionConsumer` Javadoc.

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" on page 5-29.

For more information about the `QueueConnection` class methods, see the `javax.jms.QueueConnection` Javadoc. For more information about the `ConnectionConsumer` class, see the `javax.jms.ConnectionConsumer` Javadoc.

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" on page 5-29.

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" on page 5-17.

For more information about the `TopicConnection` class methods, see the `javax.jms.TopicConnection` Javadoc. For more information about the `ConnectionConsumer` class, see the `javax.jms.ConnectionConsumer` Javadoc.

Example: Setting Up a PTP Client Server Session Pool

The following example illustrates 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 PTP Application" on page 4-14. 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:

```
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 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 in JNDI.

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

Step 2

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

The code defines the following:

- `queue` as the associated queue
- `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" on page 2-8 or the `javax.jms` Javadoc.

Example: Setting Up a Pub/Sub Client Server Session Pool

The following example illustrates 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 Pub/Sub Application" on page 4-18. It also sets up the server session pool.

The following illustrates `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 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" on page 2-8 or the `javax.jms` Javadoc.

Index

M

N

O

P

Q

R

S

T

U

X