BEAWebLogic
Server®

Programming WebLogic JMS

Version 9.2
Revised: January 23, 2007

Contents

3. Best Practices for Application Design

4. Developing a Basic JMS Application

5. Managing Your Applications

6. Using JMS Module Helper to Manage Applications

7. Using Multicasting with WebLogic Server

8. Using Distributed Destinations

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

10.Using Message Unit-of-Order

11. Using Unit-of-Work Message Groups

12. Using Transactions with WebLogic JMS

13.WebLogic JMS C API

14.Recovering from a Server Failure

A. Deprecated WebLogic JMS Features

B. FAQs: Integrating Remote JMS Providers

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 JMS Module Helper to Manage Applications," describes how to programatically create and manage JMS servers, Store-and-Forward Agents, and JMS system resources.

- Chapter 7, "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 8, "Using Distributed Destinations," describes how to use distributed destinations with WebLogic JMS.

- Chapter 9, "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 10, "Using Message Unit-of-Order," describes how to use Message Unit-of-Order to provide strict message ordering when using WebLogic JMS queues.

- Chapter 12, "Using Transactions with WebLogic JMS," describes how to use transactions with WebLogic JMS.

- Chapter 13, "WebLogic JMS C API," provides information on how to develop C programs that interoperate with WebLogic JMS.

- Chapter 14, "Recovering from a Server Failure," describes how to terminate a JMS application gracefully if a server fails and how to migrate JMS data after server failure.

- Appendix A, "Deprecated WebLogic JMS Features," describes features that have been deprecated for this release of WebLogic Server:

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* 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 Applications to WebLogic Server* is the primary source of information about deploying WebLogic Server applications.

Samples and Tutorials for the JMS Developer

In addition to this document, 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 9.2 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 9.2 Start menu.

Additional API examples for download at http://codesamples.projects.dev2dev.bea.com. 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 https://codesample.projects.dev2dev.bea.com.

New and Changed JMS Features In This Release

For a comprehensive listing of the new WebLogic JMS feature introduced in release 9.2, see New and Changed JMS Features In This Release in *Configuring and Managing WebLogic JMS*.

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" on page 2-1

- "Understanding the Messaging Models" on page 2-4

- "Value-Added Public JMS API Extensions" on page 2-7

- "Understanding the JMS API" on page 2-9

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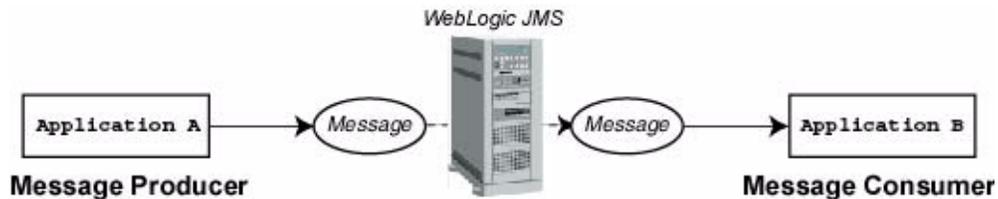
As illustrated in the figure, WebLogic JMS accepts messages from *producer* applications and delivers them to *consumer* applications.

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

The following figure illustrates the WebLogic JMS architecture.

Figure 2-2   WebLogic JMS Architecture
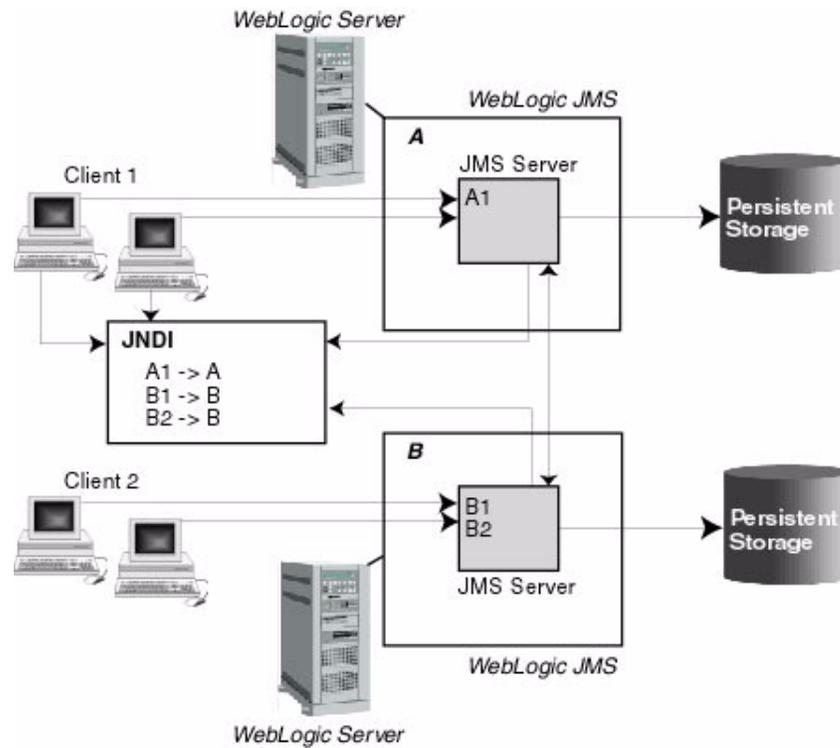
Where: A1 and B1 are connection factories and B2 is a queue.

Major Components

The major components of the WebLogic JMS Server architecture, as illustrated in Figure 2-2, include:

- JMS servers that can host a defined set of modules and any associated persistent storage that reside on a WebLogic Server instance.

- JMS modules contains configuration resources (such as queues, topics, and connections factories) 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 resource *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 2-3 Point-to-Point (PTP) Messaging
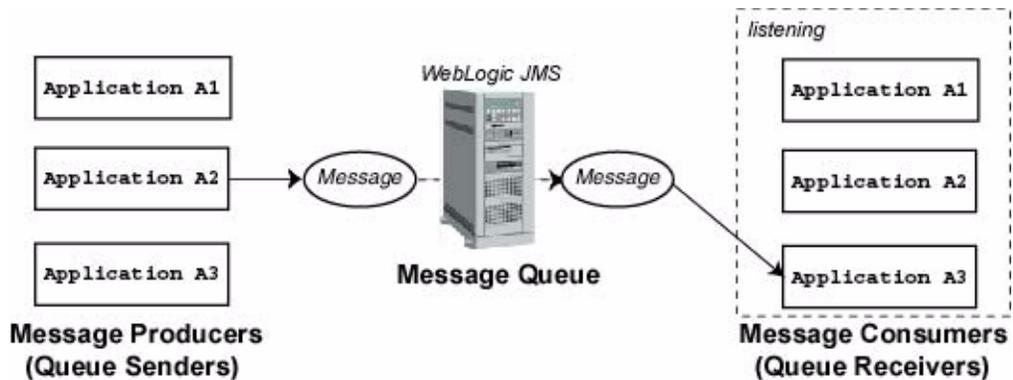
Multiple queue senders and queue receivers can be associated with a single queue, but an individual message can be delivered to only *one* queue receiver.

If multiple queue receivers are listening for messages on a queue, 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 2-4   Publish/Subscribe (Pub/Sub) Messaging
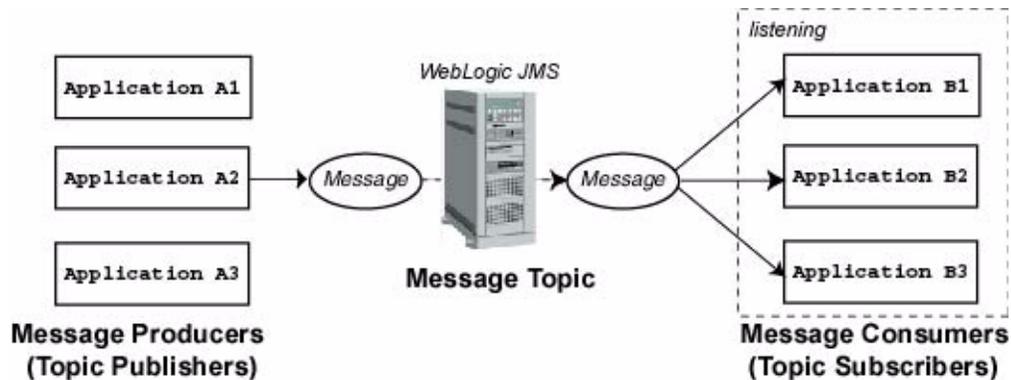
Unlike with the PTP messaging model, the pub/sub messaging model allows multiple topic subscribers to receive the same message. JMS retains the message until all topic subscribers have received it.

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 5-21.

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 and it must not be delivered twice. It is not considered sent until it has been safely written to a file or database. WebLogic JMS writes persistent messages to a WebLogic persistent store (disk-base file or JDBC-accessible database) that is optionally targeted by each JMS server during configuration.

- Non-persistent messages are not stored. They are guaranteed to be delivered *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, see "WebLogic Server Value-Added JMS Features" in *Configuring and Managing WebLogic JMS*.

WebLogic Server Value-Added JMS Features

In addition to the standard JMS APIs specified by the JMS Specification, WebLogic Server provides numerous `weblogic.jms.extensions` APIs, which includes the classes and methods described in the following table.

Table 2-1 WebLogic JMS Public API Extensions

Interface/Class	Function
ConsumerInfo, DestinationInfo	Provides consumer and destination information to management clients in CompositeData format.
JMSMessageFactoryImpl, WLMessageFactory	Provides a factory and methods to: • Create JMS messages • Create JMS bytes messages • Create JMS map messages • Creating JMS object messages • Creating JMS stream messages • Creating JMS text messages • Creating JMS XML messages
JMSMessageInfo	Provide browsing and message manipulation using JMX.
JMSModuleHelper, JMSNamedEntityModifier	Monitors JMS runtime MBeans and manages JMS Module configuration entities in a JMS module.
JMSRuntimeHelper	Monitors JMS runtime JMX MBeans.

Table 2-1 WebLogic JMS Public API Extensions

Interface/Class	Function
MDBTransaction	Associates a message delivered to a MDB (message-driven bean) with a transaction.
WLDestination	Determines if a destination is a queue or a topic.
WLMessage	Sets a delivery time for messages, redelivery limits, and send timeouts.
WLMessageProducer	Sets a message delivery times for producers and Unit-of-Order names.
WLQueueSession, WLSession, WLTopicSession	Provides additional fields and methods that are not supported by `javax.jms.QueueSession`, `javax.jms.Session`, and `javax.jms.TopicSession`.
XMLMessage	Creates XML messages.
Schedule	Sets a scheduled delivery times for messages.
JMSHelper	Monitors JMS runtime MBeans. Deprecated in this release of WebLogic Server. Replaced by JMSModuleHelper.
ServerSessionPoolFactory, ServerSessionPoolListener	Provides interfaces for creating server session pools and message listeners. **Note:** Session pool configuration objects are deprecated for 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	Description
`ConnectionFactory`	Encapsulates connection configuration information. A connection factory is used to create connections. You look up a connection factory using JNDI.
`Connection`	Represents an open communication channel to the messaging system. A connection is used to create sessions.
`Session`	Defines a serial order for the messages produced and consumed.
`Destination`	Identifies a queue or topic, encapsulating the address of a specific provider. Queue and topic destinations manage the messages delivered from the PTP and pub/sub messaging models, respectively.
`MessageProducer and MessageConsumer`	Provides the interface for sending and receiving messages. Message producers send messages to a queue or topic. Message consumers receive messages from a queue or topic.

Table 2-2 WebLogic JMS Classes

JMS Class	Description
Message	Encapsulates information to be sent or received.
ServerSessionPoolFactory[1]	Encapsulates configuration information for a server-managed pool of message consumers. The server session pool factory is used to create server session pools.
ServerSessionPool[2]	Provides a pool of server sessions that can be used to process messages concurrently for connection consumers.
ServerSession[3]	Associates a thread with a JMS session.
ConnectionConsumer[4]	Specifies a consumer that retrieves server sessions to process messages concurrently.

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

For information about configuring JMS resources, see "Configuring 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 4-2.

ConnectionFactory

A ConnectionFactory encapsulates connection configuration information, and enables JMS applications to create a Connection. 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 create a user-defined a connection factory if the settings of the default factories are not suitable for your application. The main difference between the preconfigured settings for the default connection factories is the default value for the "XA Connection Factory Enabled" attribute which is used to enable JTA transactions, as shown in the following table.

Table 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 12, "Using Transactions with WebLogic JMS."

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

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

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

For more information on enabling or disabling the default connection factories, see "Servers: Configuration: Services" in the *Administration Console Online Help*.

To deploy a connection factory on specific independent servers, on specific servers within a cluster, or on an entire cluster, you must configure a new connection factory and specify the appropriate target, as explained in "Configuring and Deploying Connection Factories" on page 2-12.

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`.

Configuring and Deploying Connection Factories

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

For information on configuring connection factories, see "Configure connection factories" in the *Administration Console Online Help*.

A system administrator establishes cluster-wide, transparent access to JMS destinations from any server in the cluster by targeting to the cluster or by targeting to one or more server instances in the cluster. This way, each connection factory can be deployed on multiple WebLogic Server instances. For more information 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.

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

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

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

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

Due to the resource overhead associated with authenticating users and setting up communications, most applications establish a single connection for all messaging. In the WebLogic Server, JMS traffic is multiplexed with other WebLogic services on the client connection to the server. No additional TCP/IP connections are created for JMS. Servlets and other server-side objects 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.

Note: For this release, you can use the JMS Version 1.1 specification connection objects or you can choose to use the subclasses.

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

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 a Connection.

WebLogic JMS Session Guidelines

The JMS 1.1 Specification allows for a generic session to have a MessageConsumer for any type of Destination object. However, WebLogic JMS does not support having both types of MessageConsumer (QueueConsumer and TopicSubscriber) for a single session. In addition, having multiple consumers for a single session is not a common practice. The following commonly-used scenarios are supported, however:

- Using a single session with both a QueueSender and a TopicSubscriber (and vice-versa: QueueConsumer and TopicPublisher).

- Multiple MessageProducers of any type.

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

Session Subclasses

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.
	For more information on the Acknowledge Policy attribute, see "JMS Connection Factory: Configuration: General" in the *Administration Console Online Help*.
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 7-1.
	Note: Use only with topics. You should avoid using this mode if your application cannot handle lost or duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.

Transacted Session

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

When you create a transacted session, the acknowledge mode is ignored. When an application commits a transaction, all the messages that the application received during the transaction are acknowledged by the messaging system and messages it sent are accepted for delivery. If an application rolls back a transaction, 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 12-4.

Destination

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

Similar to a connection factory, an administrator defines and configures the destination and 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 a distributed destination, which is a single set of destinations (queues or topics) that are accessible as a single, logical destination to a client. For more information, see "Distributed Destinations" on page 2-18.

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`.

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

The following table describes the `Destination` subclasses.

Table 2-8 Destination Subclasses

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

Table 2-8 Destination Subclasses (Continued)

Subclass	Messaging Model	Manages Messages for
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 5-26.

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

A distributed destination resource is a single set of destinations (queues or topics) that are accessible as a single, logical destination to a client (for example, a distributed topic has its own JNDI name). The members of the set are typically distributed across multiple servers within a cluster, with each member belonging to a separate JMS server. Applications that use a distributed destination are more highly available than applications that use standalone destinations because WebLogic JMS provides load balancing and failover for the members of a distributed destination in a cluster.

- For more information on using a distributed destination with your applications, see "Using Distributed Destinations" on page 8-1.

- For instructions on configuring a distributed queue destination, see "Configure uniform distributed queues" in the *Administration Console Online Help*.

- For instructions on configuring a distributed topic destination, see "Configure uniform distributed topics" in the *Administration Console Online Help*.

MessageProducer and MessageConsumer

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

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

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

Note: For this release, you can use the JMS Version 1.1 specification message producer and consumer objects or you can choose to use the subclasses.

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.
TopicPublisher	Pub/sub	Sends messages for a JMS pub/sub provider.
TopicSubscriber	Pub/sub	Receives messages for a JMS pub/sub provider.

The PTP model, as shown in the figure "Point-to-Point (PTP) Messaging" on page 2-5, 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 2-6, 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 5-21.

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 4-2, or the `javax.jms.MessageProducer` and `javax.jms.MessageConsumer` javadocs.

Message

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

- "Message Header Fields" on page 2-20

- "Message Property Fields" on page 2-25

- "Message Body" on page 2-26

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 5-26, 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 and is set directly on the message by the application before `send()`. There are two common applications for this field. The first application is to link messages by setting up a request/response scheme, as follows: 1. When an application sends a message, it stores the `JMSMessageID` value assigned to it. 2. When an application receives the message, it copies the `JMSMessageID` into the `JMSCorrelationID` field of a response message that it sends back to the sending application. The second application is to use the `JMSCorrelationID` field to carry any String you choose, enabling a series of messages to be linked with some application-determined value.	Application
JMSDeliveryMode	Specifies `PERSISTENT` or `NON_PERSISTENT` messaging. This field is set on the producer or as parameter sent by the application before `send()`. When a persistent message is sent, it is stored in the WebLogic Persistent Store. The `send()` operation is not considered successful until delivery of the message can be guaranteed. A persistent message is guaranteed to be delivered at least once. WebLogic JMS does not store non-persistent messages in the persistent store. This mode of operation provides the lowest overhead. They are guaranteed to be delivered at least once unless there is a system failure, in which case messages may be lost. If a connection is closed or recovered, all non-persistent messages that have not yet been acknowledged will be redelivered. Once a non-persistent message is acknowledged, it will not be redelivered. This value is overwritten by a call to `producer.send()`, setting this value directly on the message has no effect. The values set by the producer can be queried using the message supplied to `producer.send()` or when the message is received by a consumer.	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 is set by the application before send() and depends on timeToDeliver, which is set on the producer. This field can be used to sort messages in a destination and to select messages. For purposes of data type conversion, the JMSDeliveryTime is a long integer.	send() method
JMSDestination	Specifies the destination (queue or topic) to which the message is to be delivered. This field is set when creating producer or as parameter sent by the application before send(). This value is overwritten by a call to producer.send(), setting this value directly on the message has no effect. The values set by the producer can be queried using the message supplied to producer.send() or when the message is received by a consumer. When a message is received, its destination value must be equivalent to the value assigned when it was sent.	send() method
JMSExpiration	Specifies the expiration, or time-to-live value, for a message. This field is set by the application before send(). Depends on timeToLive, which is set on the producer or as a parameter sent by the application to send(). WebLogic JMS calculates the JMSExpiration value as the sum of the application's time-to-live and the current GMT. If the application specifies time-to-live as 0, 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. This field is set internally by send(). All JMSMessageIDs start with an ID: prefix. This value is overwritten by a call to producer.send(), setting this value directly on the message has no effect. The values set by the producer can be queried using the message supplied to producer.send() or when the message is received by a consumer. When the message is received, it contains a provider-assigned value.	send() method

Table 2-10 Message Header Fields (Continued)

Field	Description	Defined by
JMSPriority	Specifies the priority level. This field is set on the producer or as parameter sent by the application before send().	send() method
	JMS defines ten priority levels, 0 to 9, 0 being the lowest priority. Levels 0-4 indicate gradations of *normal* priority, and level 5-9 indicate gradations of *expedited* priority.	
	When the message is received, it contains the value specified by the method sending the message.	
	You can sort destinations by priority by configuring a destination key, as described in "Configure destination keys" in the *Administration Console Online Help*.	
JMSRedelivered	Specifies a flag set when a message is redelivered because no acknowledge was received. This flag is of interest to a receiving application.	WebLogic JMS
	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.	
JMSReplyTo	Specifies a queue or topic to which reply messages should be sent. This field is set directly on the message by the application before send().	Application
	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.	

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 directly on the message by the application before send(). 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.

WebLogic Server supports the use of the following JMS (JMSX) defined properties as defined in the JMS 1.1. Specification:

Table 2-11 JMSX Property

Type	Description
JMSXUserID	System generated property that identifies the user sending the message. See "Configure connection factory security parameters" in the Administration Console Online Help and Message ID Propagation Security Enhancement in *Configuring and Managing WebLogic JMS*.
JMSXDeliveryCount	System generated property that specifies the number of message delivery attempts where first attempt is 1.
JMSXGroupID	Identity of the message group.
JMSXGroupSeq	Sequence number of a message within a group.

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

- "Setting and Browsing Message Header and Property Fields" on page 5-26

- "Filtering Messages" on page 5-34

- JMS 1.1. Specification

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 2-12 JMS Message Types

Type	Description
javax.jms.BytesMessage	Stream of uninterpreted bytes, which must be understood by the sender and receiver. The access methods for this message type are stream-oriented readers and writers based on `java.io.DataInputStream` and `java.io.DataOutputStream`.
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

Note: Session pool and connection consumer 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 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 A-1, or the `weblogic.jms.extnesions.ServerSessionPoolFactory` Javadoc.

ServerSessionPool

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

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

To learn how to use the server session pool within an application, see "Defining Server Session Pools" on page A-1, or the `javax.jms.ServerSessionPool` Javadoc.

ServerSession

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

A `ServerSession` is created by a `ServerSessionPool` object.

To learn how to use the server session within an application, see "Defining Server Session Pools" on page A-1, or the `javax.jms.ServerSession` Javadoc.

ConnectionConsumer

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.

To learn how to use the connection consumers within an application, see "Defining Server Session Pools" on page A-1, or the `javax.jms.ConnectionConsumer` Javadoc.

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

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" on page 3-1
- "Message Compression" on page 3-2
- "Message Properties and Message Header Fields" on page 3-3
- "Message Ordering" on page 3-3
- "Topics vs. Queues" on page 3-4
- "Asynchronous vs. Synchronous Consumers" on page 3-4
- "Persistent vs. Non-Persistent Messages" on page 3-5
- "Deferring Acknowledges and Commits" on page 3-6
- "Using AUTO_ACK for Non-Durable Subscribers" on page 3-7
- "Alternative Qualities of Service, Multicast and No-Acknowledge" on page 3-7
- "Avoid Multi-threading" on page 3-8

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.

Server-side serialization

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

Selection

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

For information on using JMS connection factories to specify the automatic compression of messages that exceed a specified threshold size, see Tuning Weblogic JMS in the *WebLogic Performance and Tuning Guide*.

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.

For more information, see "Message Header Fields" on page 2-20 and "Message Property Fields" on page 2-25.

Message Ordering

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

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

BEA recommends applications that use Ordered Redelivery upgrade to Message Unit-of-Order. For more information, see Chapter 10, "Using Message Unit-of-Order."

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.

For more information on configuring JMS queues and topics, see Queue and Topic Destination Resources in *Configuring and Managing WebLogic JMS*.

Asynchronous vs. Synchronous Consumers

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

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

  Note: In WebLogic Server 9.2, your synchronous consumers can also use the same efficient behavior as asynchronous consumers by enabling the Prefetch Mode for Synchronous Consumers option on JMS connection factories, as described in "Using the Prefetch Mode to Create a Synchronous Message Pipeline" on page 4-30.

- 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.

For more information, see "Receiving Messages Asynchronously" on page 4-28 and "Receiving Messages Synchronously" on page 4-29.

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 using the default persistent store that is available on each targeted WebLogic Server instance

  – 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.

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.

Avoid Multi-threading

The JMS Specification states that multi-threading a session, producer, consumer, or message method results in undefined behavior except when calling `close()`. For this release, if WebLogic JMS determines that you created a multi-threaded producer, the server instance throws a `JMSException`. If your application is thread limited, try increasing the number of producers and sessions.

Developing a Basic JMS Application

The following sections provide information on the steps required to develop a basic JMS application:

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 JMS within transactions (described in "Using Transactions with WebLogic JMS" on page 12-1)

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

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.extensions	WebLogic-specific JMS public API that provides additional classes and methods, as described in "Value-Added Public JMS API Extensions" on page 2-7.
weblogic.jms.extensions.ServerSessionPoolFactory	Deprecated in WebLogic Server 8.1.

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 4-1   Setting Up a JMS Application
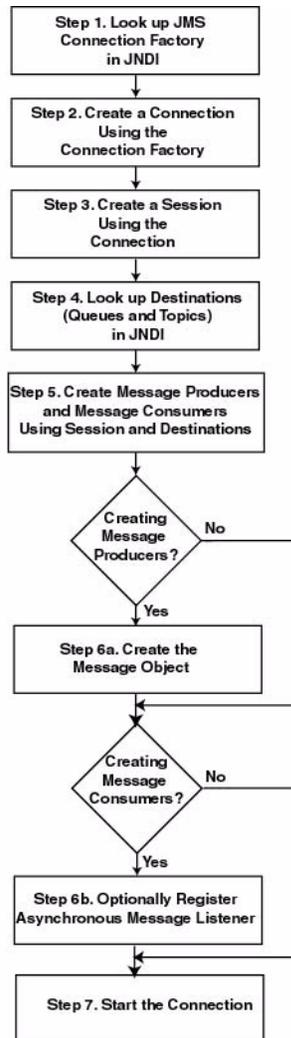
The setup steps are described in the following sections. Detailed examples of setting up a Point-to-Point (PTP) and Publish/Subscribe (Pub/Sub) application are also provided. The examples are excerpted from the `examples.jms` package provided with WebLogic Server in the `WL_HOME\samples\server\examples\src\examples\jms` 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 resources, including the connection factories, JMS servers, and destinations.

- For more information, see "Configure Messaging" in the *Administration Console Online Help*.

- For more information about the JMS classes and methods described in these sections, see "Understanding the JMS API" on page 2-9, 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 12-1.

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. For information on configuring connection factories and the defaults that are available, see "Configure connection factories" in the *Administration Console Online Help*.

Once the connection factory has been defined, you can look it up by first establishing a JNDI context (`context`) using the `InitialContext()` method. For any application other than a servlet application, you must pass an environment used to create the initial context.

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" on page 2-10 or the `javax.jms.ConnectionFactory` Javadoc.

Step 2: Create a Connection Using the Connection Factory

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

For more information about the `Connection` class, see "Connection" on page 2-13 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.

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

WebLogic JMS does not support having both types of MessageConsumer (QueueConsumer and TopicSubscriber) for a single Session. However, it does support a single session with both a QueueSender and a TopicSubscriber (and vice-versa: QueueConsumer and TopicPublisher), or with multiple MessageProducers of any type.

For more information about the `Session` class, see "Session" on page 2-14 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 2-15. 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 2-15. 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, as described in "Configure topics" and "Configure queues" in the *Administration Console Online Help*. For more information about the `Destination` class, see "Destination" on page 2-17 or the `javax.jms.Destination` Javadoc.

Once the destination has been configured, you can look up a destination using one of the following procedures:

Using a JNDI Name

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 4-4, 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.

Use a Reference

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

Note: The `createQueue()` and `createTopic()` methods ***do not create*** destinations dynamically; they create only references to destinations that already exist. For information about creating destinations dynamically, see "Using JMS Module Helper to Manage Applications" on page 6-1.

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

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

The value of `queueName` and/or `topicName` string is defined by:

- The `CreateDestinationIdentifier` when configuring a destination, see JMS Queue: Configuration: General and JMS Topic: Configuration: General. For more information about these methods, see the `javax.jms.QueueSession` and `javax.jms.TopicSession` Javadoc, respectively.

- A string is defined by `JMS_Server_Name/Module_Name!Destination_Name` (for example, `myjmsserver/myModule-jms!mydestination`). When referencing a destination in an interop module, the string is defined by `JMS_Server_Name/interop-jms!Destination_Name` (for example, `myjmsserver/interop-jms!mydestination`).

Note: To reference destination in releases prior to WebLogic 9.0, use a string defined by `JMS_Server_Name!Destination_Name` (for example, `myjmsserver!mydestination`).

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.

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" on page 2-19, 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 5-34.

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 4-20.

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 5-23. 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 5-34.

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 4-20.

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" on page 2-20, 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 5-12.

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 5-12.

The `MessageConsumer` class methods are inherited by the `QueueReceiver` and `TopicSubscriber` classes. For additional information about the `MessageConsumer` class methods, see "MessageProducer and MessageConsumer" on page 2-19 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" on page 2-14.

```
 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 2-9 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" on page 2-14.

```
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 2-9 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 4-2, you can send messages. To send a message, you must perform the following steps:

1. "Create a Message Object" on page 4-20.

2. "Define a Message" on page 4-20

3. "Send the Message to a Destination" on page 4-21

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 4-27.

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 4-11.

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 4-11. 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.

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 4-2.

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" on page 2-19 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:

- Connection factory or destination override configuration attributes defined for the producer, as described "Configure default delivery parameters" in the *Administration Console Online Help*.

- 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, as described in "Configure persistent stores" in the *Administration Console Online Help*.

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

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 4-9.

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:

- Connection factory or destination override configuration attributes defined for the producer, as described "Configure default delivery parameters" in the *Administration Console Online Help*.

- 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, as described in "Configure persistent stores" in the *Administration Console Online Help*.

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 4-10.

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, as described in "Configure connection factories" in the *Administration Console Online Help*.

Alternatively, you can set the delivery mode, priority, time-to-deliver, time-to-live, and redelivery delay (timeout), and redelivery limit values dynamically using the message producer's set methods. 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, as described in "Configure message delivery overrides: Queues" and "Configure message delivery overrides: Topics" in the *Administration Console Online Help*.

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`

Table 4-2 Message Producer Set and Get Methods (Continued)

Attribute	Set Method	Get Method
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 4-2, 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 4-20.

- Destination sort order set using destination keys, as described in "Configure destination keys" in the *Administration Console Online Help*.

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 4-20.

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 4-12.

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 per Session attribute on the client's connection factory, which is defined as the "maximum number of messages that can exist for an asynchronous consumer and that have not yet been passed to the message listener". The default setting is *10*. For more information on configuring a JMS connection factory, see "Configure connection factories" in the *Administration Console Online Help*.

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 per Session 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.

Using the Prefetch Mode to Create a Synchronous Message Pipeline

In releases prior to WebLogic Server 9.1, synchronous consumers required making a two-way network calls for each message, which was an inefficient model because the synchronous consumer could not retrieve multiple messages, and could also increase network traffic resources, since synchronous consumers would continually poll the server for available messages. In WebLogic 9.1 or later, your synchronous consumers can also use the same efficient behavior as asynchronous consumers by enabling the Prefetch Mode for Synchronous Consumers option on JMS connection factories, either using the Administration Console or the `JMSClientParamsBean` MBean.

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

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

For more information on the behavior of pipelined messages, see "Asynchronous Message Pipeline" on page 4-28. For more information on configuring a JMS connection factory, see "Configure connection factories" in the *Administration Console Online Help*.

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 2-15. 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 2-15.

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 5-16, 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.

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" on page 5-2

- "Setting a Redelivery Limit for Messages" on page 5-3

- "Ordered Redelivery of Messages" on page 5-4

- "Handling Expired Messages" on page 5-6

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.

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

The application that creates the session can then override the connection factory setting using WebLogic-specific extensions to the javax.jms.Session interface. The session attribute is dynamic and can be changed at any time. Changing the session redelivery delay affects all messages consumed and rolled back (or recovered) by that session after the change except when the message is in a session using non-durable topics.

Note: When a session is using non-durable topics, the setRedeliveryDelay method does not apply. This may result in unexpected behavior if you are using a non-durable topic consumer to drive a workflow.

The method for setting the redelivery delay on a session is provided through the weblogic.jms.extensions.WLSession interface, which is an extension to the javax.jms.Session interface. To define a redelivery delay for a session, use the following methods:

```
public void setRedeliveryDelay(
 long redeliveryDelay
) throws JMSException;

public long getRedeliveryDelay(
) throws JMSException;
```

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

Overriding the Redelivery Delay on a Destination

Regardless of what redelivery delay is set on the session, the destination where a message is being rolled back or recovered can override the setting. The redelivery delay override applied to the redelivery of a message is the one in effect at the time a message is rolled back or recovered.

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

- "Configure message delivery failure options: Queues" in the *Administration Console Online Help*

- "Configure message delivery failure options: Topics" in the *Administration Console Online Help*

Setting a Redelivery Limit for Messages

You can specify a limit on the number of times that WebLogic JMS will attempt to redeliver a message to an application. 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. For more information, see:

- "Configure message delivery failure options: Queues" in the *Administration Console Online Help*.

- "Configure message delivery failure options: Topics" in the *Administration Console Online Help*.

Configuring an Error Destination for Undelivered Messages

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

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

- "Configure message delivery failure options: Queues" in the *Administration Console Online Help*.

- "Configure message delivery failure options: Topics" in the *Administration Console Online Help*.

- "Uniform distributed topics - configure delivery failure parameters" in the *Administration Console Online Help*.

- "Uniform distributed topics - configure delivery failure parameters" in the *Administration Console Online Help*.

Ordered Redelivery of Messages

Note: BEA recommends that applications that use Ordered Redelivery upgrade to Message Unit-of-Order. See "Using Message Unit-of-Order" on page 10-1.

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 EJBs" 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.

Table 5-1 Message Delivery Schedule

Example	Description
`0 0 0,30 * * * *`	Exact next nearest half-hour
`* * 0,30 4-5 * * *`	Anytime in the first minute of the half hours 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

Table 5-1 Message Delivery Schedule

Example	Description
`* * * * 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 `l` 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.

Table 5-2 JMS Metadata

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

For more information about the `ConnectionMetaData` class, see the `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 4-2. 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 12-4.

- 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 12-4.

- 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.

Managing Destinations

The following sections describe how to create and delete destinations:

- Dynamically Creating Destinations

- Dynamically Deleting Destinations

Dynamically Creating Destinations

You can create destinations dynamically using:

- Using JMS Module Helper to Manage Applications

- Using Temporary Destinations

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

Dynamically Deleting Destinations

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

- "Using JMS Module Helper to Manage Applications" on page 6-1

- 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 8-1.

- The destination must not be the error destination for some other destination. For more information, see .

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

What Happens when a Destination is Deleted

When a destination is deleted, the following behaviors and semantics apply:

- Physical deletion of existing messages — all durable subscribers for the deleted destination are permanently deleted. All messages, persistent and non-persistent, stored in the deleted destination are permanently removed from the messaging system.

- No longer able to create producers, consumers, and browsers — 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.

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 5-21.

Because messages are never available if the server is restarted, all `PERSISTENT` messages are silently made `NON_PERSISTENT`. As a result, temporary destinations are not suitable for business logic that must survive a restart.

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

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 8-1.

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

- Best Practice: Always Close Failed JMS ClientIDs

- Deleting Durable Subscriptions

- Modifying Durable Subscriptions

- Managing Durable Subscriptions

Defining the Persistent Store

You must configure a persistent file or database store and assign it to your JMS server so WebLogic JMS can store a message until it 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.

Note: Valid durable subscription names can not include the following characters: comma "**,**", equals "**=**", colon "**:**", asterisk "*****", percent "**%**", or question mark"**?**". '

You may also specify a message selector for filtering messages and a `noLocal` flag (described later in this section). Message selectors are described in more detail in "Filtering Messages" on page 5-34. 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 5-22.

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.

Best Practice: Always Close Failed JMS ClientIDs

As a best practice, JMS clients should always call the `close()` method instead of allowing the application to rely on the JVM's garbage collection to clean up failed JMS connections. This is particularly important for durable subscription ClientIDs because the JMS Automatic Reconnect feature keeps a reference to such failed JMS connections. Therefore, always use `connection.close()` to clean up your connections. Also, consider using a `finally` block to ensure that your connection resources are cleaned up. Otherwise, WebLogic Server allocates system resources to keep the connection available.

The following snippet demonstrates using `close()` and `finally` in a JMS client to clean up failed connection resources:

```
JMSConnection con = null;
try {
  con = cf.createConnection();
  con.setClientID("Fred");
  // Do some I/O stuff;
}
finally {
  if (con != null) con.close();
}
```

For more information about the JMS Automatic Reconnect feature, see "Automatic JMS Client Failover" on page 14-2.

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 5-26.

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 5-25.

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 5-23 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 manage durable topic subscribers using either the Administration Console or through public runtime APIs. This functionality also enables you to view and browse *all* messages, and to manipulate *most* messages on durable subscribers. This includes message browsing (for sorting), message manipulation (such as move and delete), and message import and export. For more information, see "Managing JMS Messages" in *Configuring and Managing WebLogic JMS.*

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 5-34.

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.

When modifying (overriding) header field values, you need to take into consideration instances when message fields are overwritten by the JMS subsystem. For instance, setting the priority on a producer affects the priority of the message, but a value supplied to the `send()` method overrides the setting on the producer. Similarly, values set on a destination override values set by the producer or values supplied to the `send()` method. The only way to verify the value of header fields is to query the message after a `send()` method.

For a description of the standard messages header fields, see "Message Header Fields" on page 2-20.

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.

Table 5-3 JMS Header Field Methods

Header Field	Set Method	Get Method
JMSCorrelationID	public void setJMSCorrelationID(String correlationID) throws JMSException	public String getJMSCorrelationID( ) throws JMSException public byte[] getJMSCorrelationIDAsBytes( ) throws JMSException
JMSDestination[1]	public void setJMSDestination(Destination destination) throws JMSException	public Destination getJMSDestination( ) throws JMSException

Table 5-3 JMS Header Field Methods

Header Field	Set Method	Get Method

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`

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`

Table 5-3 JMS Header Field Methods

Header Field	Set Method	Get Method
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```
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 5-4 Message Property Set and Get Methods for Data Types

Data Type	Set Method	Get Method

boolean	```public void setBooleanProperty(```
	``` String name,```
	``` boolean value```
	```) throws JMSException```
		```public boolean```
		```getBooleanProperty(```
		``` String name```
		```) throws JMSException```

byte	```public void setByteProperty(```
	``` String name,```
	``` byte value```
	```) throws JMSException```
		```public byte getByteProperty(```
		``` String name```
		```) throws JMSException```

double	```public void setDoubleProperty(```
	``` String name,```
	``` double value```
	```) throws JMSException```
		```public double getDoubleProperty(```
		``` String name```
		```) throws JMSException```

Table 5-4 Message Property Set and Get Methods for Data Types (Continued)

Data Type	Set Method	Get Method
float	```public void setFloatProperty(String name, float value) throws JMSException```	```public float getFloatProperty(String name) throws JMSException```
int	```public void setIntProperty(String name, int value) throws JMSException```	```public int getIntProperty(String name) throws JMSException```
long	```public void setLongProperty(String name, long value) throws JMSException```	```public long getLongProperty(String name) throws JMSException```
short	```public void setShortProperty(String name, short value) throws JMSException```	```public short getShortProperty(String name) throws JMSException```
String	```public void setStringProperty(String name, String value) throws JMSException```	```public String getStringProperty(String name) throws JMSException```

In addition to the set and get methods described in the previous table, you can use the `setObjectProperty()` and `getObjectProperty()` methods to use the objectified primitive values of the property type. When the objectified value is used, the property type can be determined at execution time rather than during the compilation. The valid object types are boolean, byte, double, float, int, long, short, and string.

You can access all property field names using the following Message method:

```
public Enumeration getPropertyNames(
) throws JMSException
```

This method returns all property field names as an enumeration. You can then retrieve the value of each property field by passing the property field name to the appropriate get method, as described in the 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 5-5 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
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 2-25 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 5-34.

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 5-26 and "Setting Message Property Fields" on page 5-30, 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.

Sending XML Messages

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

The WebLogic Server JMS API 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 5-39

- Using a String Representation

- Using a DOM Representation

WebLogic XML APIs

You can use the following WebLogic XML APIs for transformation of XML between `String` and `DOM` representations:

- XMLMessage-Use to send messages with XML content.

- `WLSession.createXMLMessage`- Use to create an XML message.

It is possible for the payload of `XMLMessage` to be set using one XML representation and retrieved using a different representation. For example, it is valid for the XMLMessage body to be set using a `String` representation and be retrieved using a DOM representation.

Using a String Representation

Use the following steps to publish an XML message using a `string` type:

1. Serialize the XML to a `StringWriter`.

2. Call `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.

Using JMS Module Helper to Manage Applications

The `weblogic.jms.extensions.JMSModuleHelper` class contains APIs that you can use to programmatically create and manage JMS servers, Store-and-Forward Agents, and JMS system resources.

- "Configuring JMS System Resources Using JMSModuleHelper" on page 6-1

- "Configuring JMS Servers and Store-and-Forward Agents" on page 6-2

- "JMSModuleHelper Sample Code" on page 6-2

- "Best Practices when Using JMSModuleHelper" on page 6-6

Configuring JMS System Resources Using JMSModuleHelper

`JMSModuleHelper` provides the following API signatures to manage a system module and JMS resources, such as queues and topics:

- Create a resource

- Create and modify resource

- Delete a resource

- Find and modify a resource

- Find using a template

You can manage a system module, including the JMS resources it contains by providing the domain MBean or by providing the initial context to the administration server in the API signature. For more information on JMS system resources, see "Configuring JMS System Resources" in the *Administration Console Online Help*.

Configuring JMS Servers and Store-and-Forward Agents

`JMSModuleHelper` provides the following method APIs to manage JMS servers and Store-and-Forward Agents:

- Create JMS servers and Store-and-Forward Agents

- Delete JMS servers and Store-and-Forward Agents

- Deploy JMS servers and Store-and-Forward Agents

- Undelploy JMS servers and Store-and-Forward Agents

You can manage JMS servers and Store-and-Forward Agents by providing the domain MBean or by providing the initial context to the administration server in the API signature. For more information, see:

- "Configuring JMS System Resources" in the *Administration Console Help*.

- "Understanding the Store-and-Forward Service" in the *Administration Console Help*.

JMSModuleHelper Sample Code

This section provides sample code to create and delete a JMS system resource module.

Creating a JMS System Resource

The module contains a connection factory and a topic.

Listing 6-1   Create JMS System Resources

```
.
.
.
private static void createJMSUsingJMSModuleHelper(Context ctx){
System.out.println(
```

```
                 "\n\n.... Configure JMS Resource for C API Topic Example ....\n\n");

     try {

     MBeanHome mbeanHome =
          (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
          DomainMBean domainMBean = mbeanHome.getActiveDomain();
          String domainMBeanName = domainMBean.getName();
          ServerMBean[] servers = domainMBean.getServers();

      String jmsServerName = "examplesJMSServer";

//
// create a JMSSystemResource "CapiTopic-jms"
//
     String resourceName = "CapiTopic-jms";
     JMSModuleHelper.createJMSSystemResource(
         ctx,
          resourceName,
          servers[0].getName());
          JMSSystemResourceMBean jmsSR =
               JMSModuleHelper.findJMSSystemResource(
                    ctx,
                    resourceName);
           JMSBean jmsBean = jmsSR.getJMSResource();
     System.out.println("Created JMSSystemResource " + resourceName);

//
// create a JMSConnectionFactory "CConFac"
//
     String factoryName = "CConFac";
     String jndiName = "CConFac";
     JMSModuleHelper.createConnectionFactory(
         ctx,
         resourceName,
          factoryName,
          jndiName,
           servers[0].getName());
```

```
    JMSConnectionFactoryBean factory =
        jmsBean.lookupConnectionFactory(factoryName);
    System.out.println("Created Factory " + factory.getName());

//
// create a topic "CTopic"
//
    String topicName = "CTopic";
    String topicjndiName = "CTopic";
    JMSModuleHelper.createTopic(
        ctx,
        resourceName,
        jmsServerName,
        topicName,
        topicjndiName);

    TopicBean topic = jmsBean.lookupTopic(topicName);
    System.out.println("Created Topic " + topic.getName());
    } catch (Exception e) {
        System.out.println("Example configuration failed :" + e.getMessag
e());
        e.printStackTrace();
    }
}
.
.
.
```

Deleting a JMS System Resource

The following code removes JMS system resources.

Listing 6-2   Delete JMS System Resources

```
.
.
```

```
.
private static void deleteJMSUsingJMSModuleHelper(Context ctx ) {

    System.out.println("\n\n.... Remove JMS System Resource for C API Top
ic Example ....\n\n");


    try {

        MBeanHome mbeanHome =
            (MBeanHome) ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
            DomainMBean domainMBean = mbeanHome.getActiveDomain();
            String domainMBeanName = domainMBean.getName();
             ServerMBean[] servers = domainMBean.getServers();

    String jmsServerName = "examplesJMSServer";


//
// delete JMSSystemResource "CapiTopic-jms"
//
    String resourceName = "CapiTopic-jms";
    JMSModuleHelper.deleteJMSSystemResource(
        ctx,
        resourceName
         );
    } catch (Exception e) {
        System.out.println("Example configuration failed
:" + e.getMessage());
        e.printStackTrace();
    }
}
.
.
.
```

Best Practices when Using JMSModuleHelper

This section provides best practices information when using JMSModuleHelper to configure JMS servers and resources:

- Trap for Null MBean objects (such as servers, JMS servers, modules) before trying to manipulate the MBean object.

- A create or delete method call can fail without throwing an exception. In addition, a thrown exception does not necessarily indicate that the method call failed.

- The time required to create the destination on the JMS server and propagate the information to the JNDI namespace can be significant. The propagation delay increases if the environment contains multiple servers. It is recommended that you test for the existence of the queue or topic, respectively, using the session `createQueue()` or `createTopic()` method, rather than perform a JNDI lookup. By doing so, you can avoid some of the propagation-specific delay.

  For example, the following method, `findQueue()`, attempts to access a dynamically created queue, and if unsuccessful, sleeps for a specified interval before retrying. A maximum retry count is established to prevent an infinite loop.

```
private static Queue findQueue (
 QueueSession queueSession,
 String jmsServerName,
 String queueName,
 int retryCount,
 long retryInterval
) throws JMSException
{
  String wlsQueueName = jmsServerName + "/" + queueName;
  String command = "QueueSession.createQueue(" +
       wlsQueueName + ")";
  long startTimeMillis = System.currentTimeMillis();
  for (int i=retryCount; i>=0; i--) {
      try {
       System.out.println("Trying " + command);
       Queue queue = queueSession.createQueue(wlsQueueName);
       System.out.println(command + "succeeded after " +
             (retryCount - i + 1) + " tries in " +
             (System.currentTimeMillis() - startTimeMillis) +
```

```
            " millis.");
         return queue;
        } catch (JMSException je) {
         if (retryCount == 0) throw je;
        }
        try {
         System.out.println(command + "> failed, pausing " +
                retryInterval + " millis.");
         Thread.sleep(retryInterval);
        } catch (InterruptedException ignore) {}
  }
  throw new JMSException("out of retries");
}
```

You can then call the `findQueue()` method after the `JMSModuleHelper` class method call to retrieve the dynamically created queue once it becomes available. For example:

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

Using JMS Module Helper to Manage Applications

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 Multicasting

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 7-1 Setting Up Multicasting
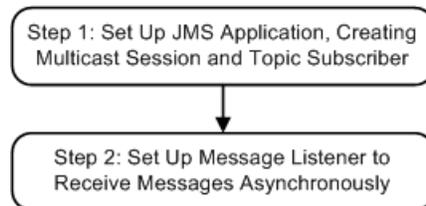
Note: Multicasting is only supported for the Pub/Sub messaging model, and only for non-durable subscribers.

Monitoring statistics are not provided for multicast sessions or consumers.

Prerequisites for Multicasting

Before setting up multicasting, the connection factory and destination must be configured to support multicasting, as follows:

- For each connection factory, the system administrator configures the maximum number of outstanding messages that can exist on a multicast session and whether the most recent or oldest messages are discarded in the event the maximum is reached. If the message maximum is reached, a `DataOverrunException` is thrown, and messages are automatically discarded. These attributes are also dynamically configurable, as described in "Dynamically Configuring Multicasting Configuration Attributes" on page 7-5.

- 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 7-5.

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.

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

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 4-2. However, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-6, 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 4-10.

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 7-2   Multicasting Sequence Gap
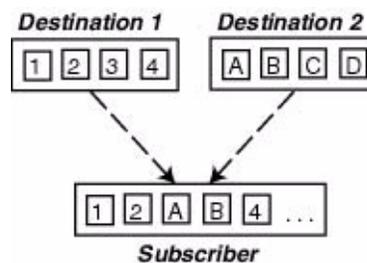
Upon receiving the "4" message from Destination 1, a `SequenceGapException` is thrown to notify the application that a message was received out of sequence. If subsequently received, the "3" message will be discarded.

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 7-1 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 "Configure topic multicast parameters" 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.

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

Consider the following network diagram.

Figure 7-3 Multicast TTL Example

TTL Count

Subnet A

Multicast Pubslisher

Router   0

Subnet B

Router   1

Multicast Subscriber

Subnet C

Router   2

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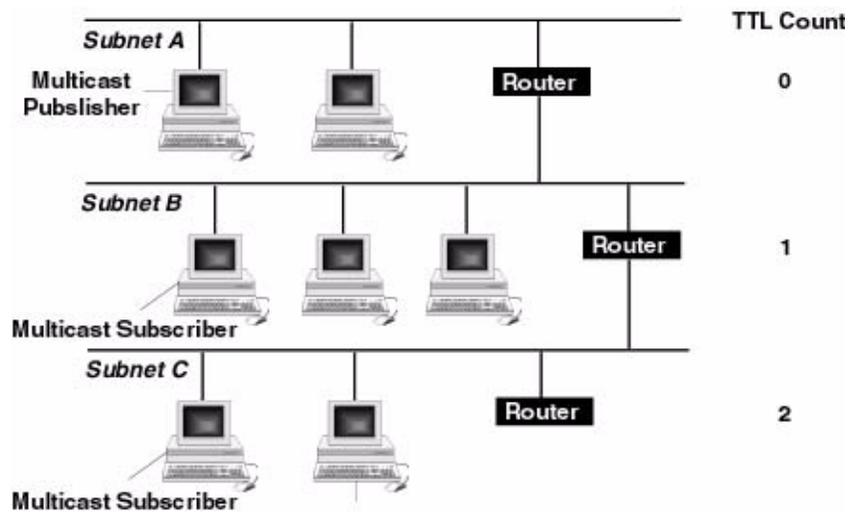
In the figure, the network consists of three subnets: Subnet A containing the multicast publisher, and Subnets B and C each containing one multicast subscriber.

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.

Using Distributed Destinations

The following sections describe the concepts and functionality of distributed destinations necessary to design higher availability applications:

- "What is a Distributed Destination?" on page 8-1

- "Why Use a Distributed Destination" on page 8-2

- "Creating a Distributed Destination" on page 8-2

- "Types of Distributed Destinations" on page 8-2

- "Using Distributed Destinations" on page 8-3

- "Using Message-Driven Beans with Distributed Destinations" on page 8-10

- "Common Use Cases for Distributed Destinations" on page 8-10

What is a Distributed Destination?

A *distributed* destination is a set of destinations (queues or topics) that are accessible as a single, logical destination to a client. A distributed destination has the following characteristics:

- It is referenced by its own JNDI name.

- Members of the set are usually distributed across multiple servers within a cluster, with each destination member belonging to a separate JMS server.

Why Use a Distributed Destination

Applications that use distributed destinations are more highly available than applications that use simple destinations because WebLogic JMS provides load balancing and failover for member destinations of a distributed destination within a cluster. Once properly configured, your producers and consumers are able to send and receive messages through the distributed destination. WebLogic JMS then balances the messaging load across all available members of the distributed destination. When one member becomes unavailable due a server failure, traffic is then redirected toward other available destination members in the set. For more information on how destination members are load balanced, see "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

Creating a Distributed Destination

Distributed destinations are created by the system administrator using the Administration Console. For more information, see "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

Types of Distributed Destinations

WebLogic Server supports two types of distributed destinations:

- "Uniform Distributed Destinations" on page 8-2
- "Weighted Distributed Destinations" on page 8-3

Uniform Distributed Destinations

In a uniform distributed destination (UDD), each of the member destinations has a consistent configuration of all distributed destination parameters, particularly in regards to weighting, security, persistence, paging, and quotas.

BEA recommends using UDDs because you no longer need to create or designate destination members, but instead rely on WebLogic Server to uniformly create the necessary members on the JMS servers to which a UDD is targeted. This feature of UDDs provides dynamic updating of a UDD when a new member is added or a member is removed.

For example, if UDD is targeted to a cluster, there is a UDD member on every JMS server in the cluster. If a new JMS server is added, a new UDD member is dynamically added to the UDD. Likewise, if a JMS server is removed, the corresponding UDD member is removed from the

UDD. This allows UDDs to provide higher availability by eliminating bottlenecks caused by configuration errors. For more information, see "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

Weighted Distributed Destinations

In a weighted distributed destination, the member destinations do not have a consistent configuration of all distributed destination parameters, particularly in regards to weighting, security, persistence, paging, and quotas.

BEA recommends converting weighted distributed destinations to UDDs because of the administrative inflexibility when creating members that are intended to carry extra message load or have extra capacity (more weight). Lack of a consistent member configuration can lead to unforeseen administrative and application problems because the weighted distributed destination can not be deployed consistently across a cluster.

For more information, see "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

Using Distributed Destinations

A distributed destination is a set of physical JMS destination members (queues or topics) that is accessed through a single JNDI name. As such, a distributed destination can be looked up using JNDI. 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` preceded by the parent module name, separated by an exclamation point (!). No JMS server name or separating forward slash (/) is required.

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

```
topic = myTopicSession.createTopic("myModuel!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 the name of a distributed destination. If no such destination member exists, then the call will fail with an `InvalidDestinationException`. See "Deploying Message-Driven Beans on a Distributed Topic" on page 8-8

Using Distributed Queues

A distributed queue is a set of physical JMS queue members. As such, a distributed queue can be used to create a `QueueSender`, `QueueReceiver`, and a `QueueBrowser`. The fact that a distributed queue represents multiple physical queues is mostly transparent to your application.

The queue members can be located anywhere, but must all be served by JMS servers in a single server cluster. When a message is sent to a distributed queue, it is sent to 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. '

This section provides information on using distributed queues:

- "Queue Forwarding" on page 8-4
- "QueueSenders" on page 8-4
- "QueueReceivers" on page 8-5
- "QueueBrowsers" on page 8-5

Queue Forwarding

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 "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

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 "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

QueueBrowsers

When creating a queue browser, if the supplied queue is a distributed queue, then a single physical queue member is chosen for the browser at creation time. The created queue browser is pinned to that queue member until the receiver loses its access to the queue member. At that point, any calls to the queue browser will receive a `JMSException`. Any calls to the enumeration will return a `NoSuchElementException`.

Note: The queue browser can only browse the queue member that it is pinned to. Even though a distributed queue was specified at creation time, the queue browser cannot see or browse messages for the other queue members in the distributed destination.

Using 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 8-8.

This section provides information on using distributed queues:

- "TopicPublishers" on page 8-6

- "TopicSubscribers" on page 8-7

- "Deploying Message-Driven Beans on a Distributed Topic" on page 8-8

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 "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

- 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.

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.

Accessing Distributed Destination Members

The following sections provide information on how to directly access a member a of a distributed destination.

Note: Applications defeat load balancing by directly accessing the individual physical destinations. For more information, see "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

Accessing Uniform Destination Members

In order to access a uniform destination member within a uniform distributed destination, you must look up the JNDI name or the member name using the `weblogic.jms.extensions.JMSModuleHelper` class uddMemberName and uddMemberJNDIName APIs. You can then use the JNDI name or supply the module name

followed by an exclamation point (!), the JMS server name followed by a forward slash (/), and the member name.

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

```
queue = myQueueSession.createQueue("myModule!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 distributed destination. If no such destination member exists, then the call will fail with an `InvalidDestinationException`.

Accessing Weighted Destination Members

In order to access a weighted destination member within a distributed destination, you must look up the destination member using the configured JNDI name, or supply the module name followed by an exclamation point (!), the JMS server name followed by a forward slash (/), and the `JMSQueueMBean` or `JMSTopicMBean` configuration MBean name.

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

```
queue = myQueueSession.createQueue("myModule!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 distributed destination. If no such destination member exists, then the call will fail with an `InvalidDestinationException`.

Distributed Destination Failover

Note: 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 example, a WebLogic cluster contains WLSServer1, WLSServer2, and WLSServer3 and you are connected to WLServer2. If server WLSServer 2 fails, WebLogic JMS fail the producer over to one of the remaining cluster members. For more information, see "Configuring Distributed Destinations" in *Configuring and Managing WebLogic JMS*.

A simple way to failover a client connected to a failed distributed destination is to write reconnect logic in the client code to connect to the distributed destination after catching `onException`.

Using Message-Driven Beans with Distributed Destinations

A message-driven bean (MDB) acts as a JMS message listener, which is similar to an event listener except that it receives messages instead of events. For more information on MDBs, see:

- MDBs and Messaging Models in Programming WebLogic Enterprise JavaBeans
- MDB Deployment Options in Programming WebLogic Enterprise JavaBeans.

Common Use Cases for Distributed Destinations

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

- "Maximizing Production" on page 8-10
- "Maximizing Availability" on page 8-11
- "Stuck Messages" on page 8-13

Maximizing Production

To maximize message production, Bea recommends that each member of a distributed destination be associated with a producer and a consumer. The following diagram demonstrates how to efficiently provide maximum message production and high availability using a UDD without using load balancing:

Figure 8-1   Paired Producers and Consumers
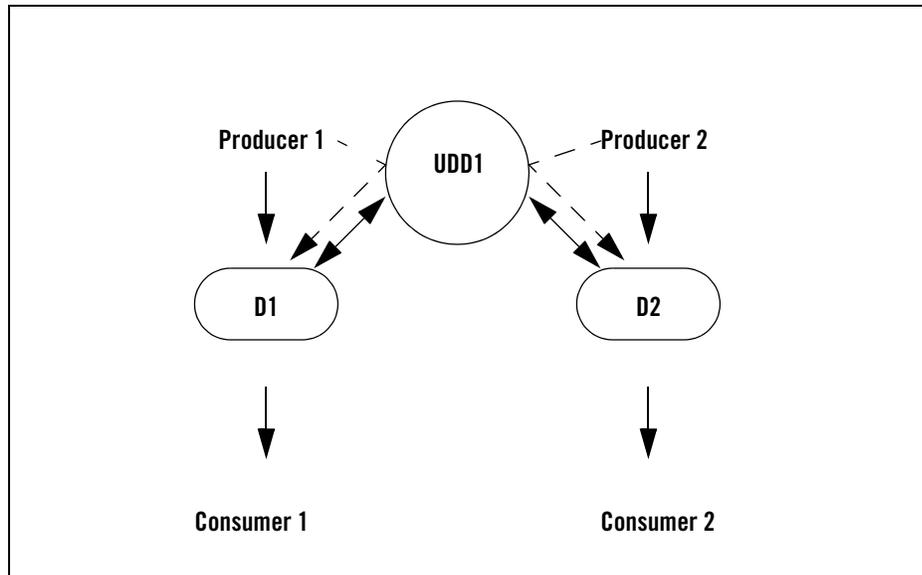
In this situation, UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. Each physical destination has a producer/consumer pair and the effective path for a message follows the solid line from the producer through the destination member to the consumer. If you are using ordering, you should have a producer for each expected Unit-of-Order. See Using Unit-of-Order with Distributed Destinations in *Programming WebLogic JMS*.

Maximizing Availability

This section provides information on how to maximize message availability.

Using Queues

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

balances an incoming message to the next available consumer using the assigned load balancing algorithm.

Note: Some combinations of Unit-of-Order features can result in the starvation of competing Unit-of-Order message streams, including the under utilization of resources when the number of consumers exceed the number of in-flight messages with different Unit-of-Order names. You will need to test your applications under maximum loads to optimize your system's performance and eliminate conditions that under utilize resources.

Using Topics

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

Figure 8-2 Using Distributed Topics
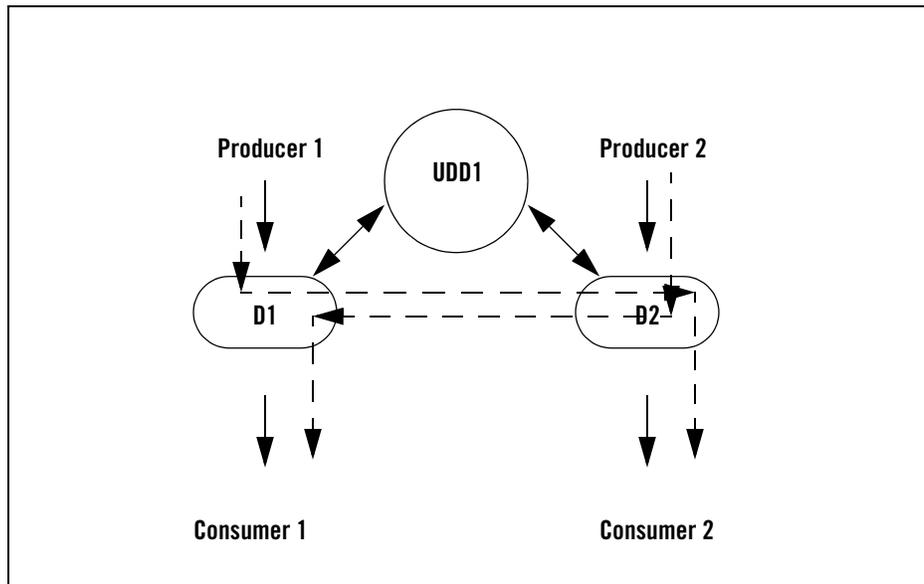
In this situation, UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. Each physical destination has a producer/consumer pair. Each consumer receives messages sent by Producer 1 and Producer 2.

Stuck Messages

In this situation, a producer is sending messages to one member of a UDD but there is no consumer available to get the message. This typically happens as a producer sends a message to one of the destinations (D1) and a consumer is listening for messages on another destination (D2).

Figure 8-3 Stuck Messages
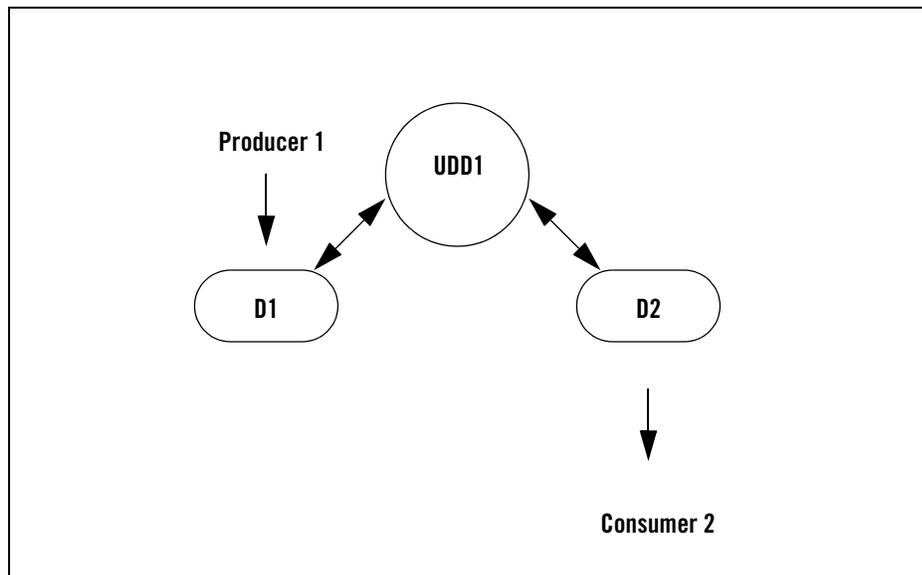
UDD1 is a uniform distributed destination composed of two physical members: D1 and D2. D1 has a producer and D2 has a consumer. Avoid this configuration by using producer/consumer pairs or by configuring forwarding on the destination.

Using Distributed Destinations

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.

- "Enabling WebLogic JMS Wrappers" on page 9-1

- "What's Happening Under the JMS Wrapper Covers" on page 9-5

- "Improving Performance Through Pooling" on page 9-8

- "Examples of JMS Wrapper Functions" on page 9-10

- "Simplified Access to Remote or Foreign JMS Providers" on page 9-15

"Simplified Access to Remote or Foreign JMS Providers" on page 9-15 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 9-5 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 9-2. 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 9-8.

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/jms/QCF
```

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

In addition to this element, there must be a matching `resource-description` element in the `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/jms/TESTQUEUE
```

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

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

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

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 9-8.

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 9-4, 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 9-15). 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 9-6; 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 EJBs" 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 9-4 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 9-2.

Monitoring Pooled Connections

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

Improving Performance Through Pooling

The automatic pooling of connections and other objects by the JMS wrappers means that it is efficient to write code as shown in "Sending a JMS Message In a J2EE Container" on page 9-4. 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 9-13 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).

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
<?xml version="1.0"?>
...

<ejb-jar>
 <enterprise-beans>
 <session>
  <ejb-name>PoolTestBean</ejb-name>
  <home>weblogic.jms.pool.test.PoolTestHome</home>
  <remote>weblogic.jms.pool.test.PoolTest</remote>
  <ejb-class>weblogic.jms.pool.test.PoolTestBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>

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

```
    </resource-ref>

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

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

</ejb-jar>
```

weblogic-ejb-jar.xml

This section declares matching `resource-description` queue connection factory and queue destination elements that tell the J2EE container which JMS connection factory and destination to put in that location.

```
<!DOC<weblogic-ejb-jar xmlns="http://www.bea.com/ns/weblogic/920"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/ns/weblogic/920
http://www.bea.com/ns/weblogic/920/weblogic-ejb-jar.xsd">

...

<weblogic-ejb-jar>
 <weblogic-enterprise-bean>
  <ejb-name>PoolTestBean</ejb-name>
  <stateless-session-descriptor>
   <pool>
    <max-beans-in-free-pool>8</max-beans-in-free-pool>
```

```
      <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
     </pool>
    </stateless-session-descriptor>

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

PoolTest.java

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

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

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

PoolTestHome.java

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

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

import java.rmi.*;
import javax.ejb.*;
```

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

PoolTestBean.java

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

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

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

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

 public void ejbActivate()
 {
 }

 public void ejbRemove()
 {
 }

 public void ejbPassivate()
```

```
    {
    }

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

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

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

    public String sendXATransactional(String text)
     throws RemoteException
    {
     String id = "Not sent yet";
     try {
```

```
   if ((qcf == null) || (destination == null)) {
    lookupJNDIObjects();
   }
   QueueConnection connection = qcf.createQueueConnection();
   try {
    QueueSession session = connection.createQueueSession
                           (false, 0);
    TextMessage message = session.createTextMessage
                           (text);
    QueueSender sender = session.createSender(destination);
    sender.send(message);
    id = message.getJMSMessageID();
   } finally {
    connection.close();
   }
  } catch (Exception e) {
   // Invalidate the JNDI objects if there is a failure
   // this is necessary because the destination object
   // 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 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 9-8 describes some ways to improve the performance of these remote lookups.

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" on page 10-2

- "Message Processing with Unit-of-Order" on page 10-2

- "Message Delivery with Unit-of-Order" on page 10-3

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:

- Many consumers on one queue. See "Using Distributed Destinations" on page 8-1.

- Multiple producers within a single application acting as a single producer. See "Using Distributed Destinations" on page 8-1.

- 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 10-11.

- Use of filters and destination sort keys. See "Message Unit-of-Order Advanced Topics" on page 10-10.

Message Processing with Unit-of-Order

The WebLogic Server Unit-of-Order feature enables a message producer or group of message producers acting as one, to group messages into a single unit that is processed sequentially in the order the messages were created. The message processing of a single message is complete when a message is acknowledged, committed, recovered, or rolled back. Until message processing for a message is complete, the remaining unprocessed messages for that Unit-of-Order are blocked.

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

- No messages from a Unit-of-Order are processed in parallel when the acknowledgement mode is `CLIENT_ACKNOWLEDGE`, `AUTO_ACKNOWLEDGE`, or `DUPS_OK_ACKNOWLEDGE`.

- When the consumer is closed, the current message processing is completed, regardless of the session's acknowledge mode.

- `CLIENT_ACKNOWLEDGE` – The application calling `Message.acknowledge` and `Session.recover` indicate which messages are completely processed in the Unit-of-Order.

- `AUTO_ACKNOWLEDGE` – The session automatically acknowledges a client's receipt of a message when it has either successfully returned from a call to `receive` or when the `MessageListener` that was called returns successfully.

  - Asynchronous mode: Successful completion or exception of `onMessage(msg)` indicates when a message is completely processed.

  - Synchronous mode: For a given consumer, such as consumer A, `consumerA.receive` is completed when one of the following occurs: `consumerA.receive`, `consumerA.setMessageListener,` or `consumerA.close`.

- `DUPS_OK_ACKNOWLEDGE` – The session automatically acknowledges a client's receipt of a message when it has either successfully returned from a call to `receive` or when the `MessageListener` that was called returns successfully.

  - Asynchronous mode: Successful completion or exception of `onMessage(msg)` indicates when a message is completely processed.

  - Synchronous mode: For a given consumer, such as consumer A, `consumerA.receive()` is completed when one of the following occurs: `consumerA.receive(), consumerA.setMessageListener(),` or `consumerA.close()`.

- `NO_ACKNOWLEDGE` – The session provides no order processing guarantees. Messages can be processed in parallel to different available consumers.

Message Delivery with Unit-of-Order

Message Unit-of-Order provides that messages are delivered in accordance with the following rules:

- Member messages of a Unit-of-Order are delivered to queue consumers sequentially in the order they were created. The message order within a Unit-of-Order will not be affected by sort criteria, priority, or filters. However, messages that are uncommitted, have a `Redelivery Delay`, or have an unexpired `TimetoDeliver` timer will delay messages that arrive after them.

- Unit-of-Order messages are processed one at a time. The processing completion of one message allows the next message in the Unit-of-Order to be delivered.

- Unit-of-Order messages sent to a distributed queue reside on only one physical member of the distributed queue. For more information, see "Using Unit-of-Order with Distributed Destinations" on page 10-12.

- All uncommitted or unacknowledged messages from the same Unit-of-Order must be in the same transaction, or if non-transactional, the same `JMSSession`. When one message in the Unit-of-Order is uncommitted or unacknowledged, the other messages are deliverable only to the same transaction or `JMSSession`. This keeps all unacknowledged messages from the same Unit-of-Order in one recoverable operation and allows order to be maintained despite rollbacks or recoveries.

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

  For Example, when Messages M_1 through M_n are delivered:

  - as part of a transaction and the transaction is rolled back (processing is complete). Then messages M_1 through M_n are delivered to any available consumer.

  - 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} is delivered to any available consumer.

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.

-

-

-

Joe Orders a Book

XYZ Online Bookstore implements a simple processing design that uses JMS to process customer orders. The JMS processing system is composed of:

- A message producer sending to a queue (Queue1).

- Multiple message driven beans (MDBs), such as MdbX and MdbY, that process messages from Queue1.

- A database (myDB) that contains order and order status information.

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

What Happened to Joe's Order

In Joe's ordering scenario, his cancel order message was processed before his purchase order message. The result was that Joe received a book he did not wish to purchase. The following steps demonstrate how Joe's order was processed.

The following diagram and corresponding actions demonstrate how Joe's order was processed.

Figure 10-1   Workflow for Joe's Order
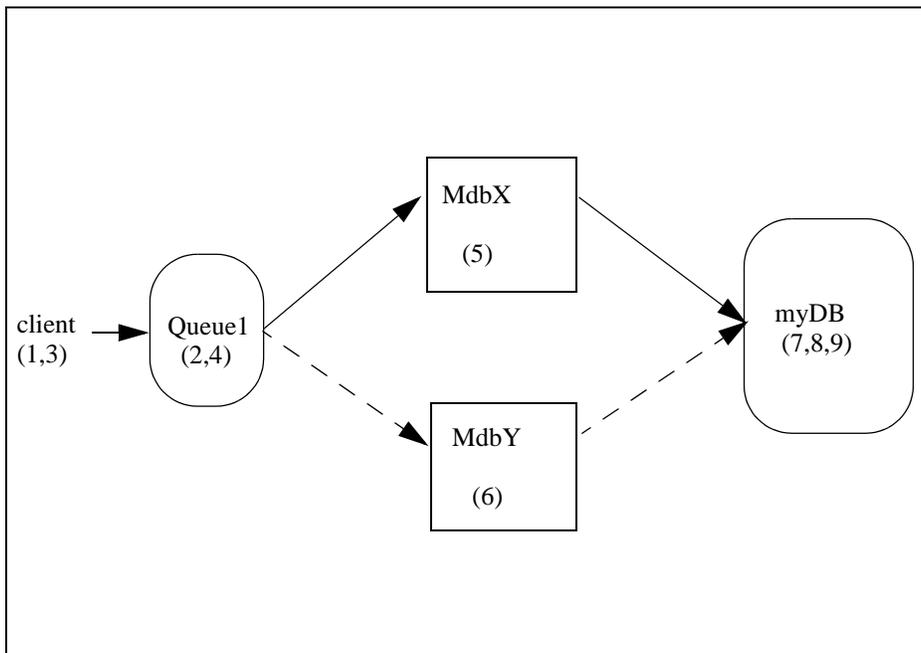
1. Joe clicks the order button from his shopping cart.

2. The order message (message A) is placed on **Queue1**.

3. Joe cancels the order.

4. The cancel order (message B) is placed on **Queue1**.

5. **MdbX** takes message A from **Queue1**.

6. **MdbY** takes message B from **Queue1**.

7. **MdbY** writes the cancel message to the database. Because there is no corresponding order message, there is no order message to remove from the database.

8. **MdbX** writes the order message to the database.

9. An application responsible for shipping books reads the database, sees the order message, and initiates shipment to Joe's home.

Although the Java Message Service Specification provides an ordered message delivery, it only provides ordered message delivery between a single instance of a producer and a single instance of a consumer. In Joe's case, multiple MDBs where available to consume messages from Queue1 and the processing order of the messages was no longer guaranteed.

How Message Unit-of-Order Solves the Problem

To ensure that all messages in Joe's order are processed correctly, the system administrator for XYZ Bookstore configures a Message Unit-of-Order based on a user session, such that all messages from a user session have a Unit-of-Order name attribute with the value of the session id. See "How to Create a Unit-of-Order" on page 10-8. All messages created during Joe's user session are processed sequentially in the order they were created because WebLogic Server guarantees that messages in a Unit-of-Order are not processed in parallel.

The following diagram and corresponding actions demonstrate how Joe's order was processed using Message Unit-of-Order.

Figure 10-2   Workflow for Joe's Order Using Unit-of-Order
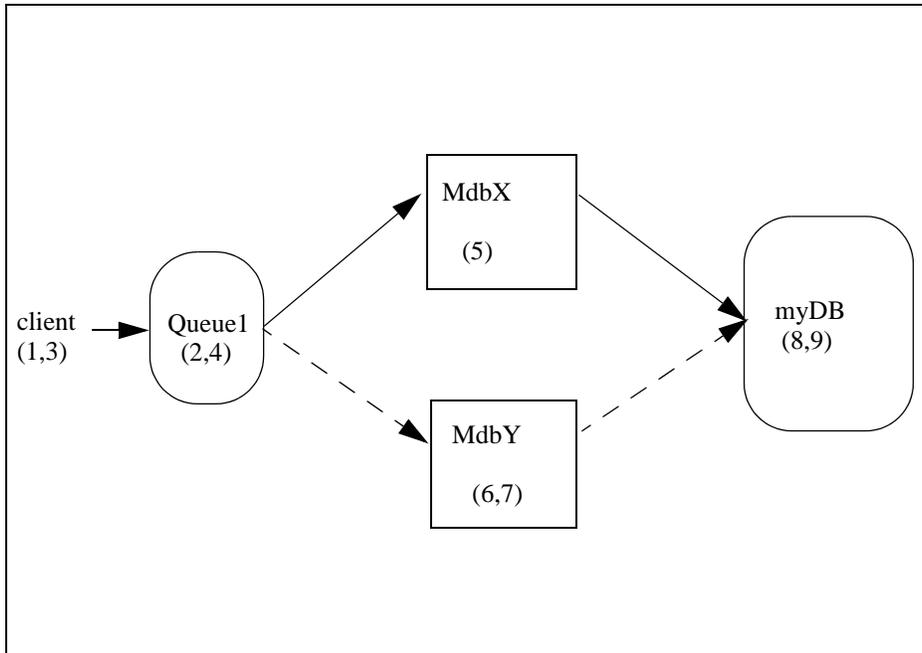
1. Joe clicks the order button from his shopping cart.

2. The order message (message A) is placed on **Queue1**.

3. Joe cancels the order.

4. The cancel order (message B) is placed on **Queue1**.

5. **MdbX** takes message A from Queue1.

6. **MdbY** takes message B from Queue1.

7. Message B on **MdbY** is blocked until **MdbX** acknowledges the order message. See "What Happens When a Message Is Delayed During Processing?" on page 10-11.

8. Message A is committed and written to the database.

9. Message B is committed and written to the database.

Because there is a corresponding order message, Joe's order is removed from the database and he does not receive a book.

How to Create a Unit-of-Order

The following sections describe how to create a Message Unit-of-Order. Also see "Message Delivery with Unit-of-Order" on page 10-3 and "Message Unit-of-Order Advanced Topics" on page 10-10.

- "Creating a Unit-of-Order Programmatically" on page 10-8
- "Creating a Unit-of-Order Administratively" on page 10-9
- "Unit-of-Order Naming Rules" on page 10-10

Creating a Unit-of-Order Programmatically

Use the `setUnitOfOrder()` method of the WLMessageProducer interface to associate a producer with a Unit-of-Order name.

For example:

```
getProducer().setUnitOfOrder("myUOOname");
```

The Unit-of-Order name attribute value is set to *myUOOname*.

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 10-1   Using the WLMessageProducer Interface to Create a Unit-of-Order

```
.
.
.
queue = (Queue)(ctx.lookup(destName));
qsender = (WLMessageProducer) qs.createProducer(queue);
qsender.setUnitOfOrder();
uooname = qsender.getUnitOfOrder();
System.out.println("Using UnitOfOrder :" + uooname);
```

.
.
.

Creating a Unit-of-Order Administratively

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

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

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

- Configure a connection factory to always use a user-generated Unit-of-Order name. As a result, all producers created from such a connection factory have Unit-of-Order enabled. See Configure connection factory unit-of-order parameters in the *Administration Console Help*.

- Configure a connection factory to always use a system-generated Unit-of-Order name for each session. See Configure unit-of-order parameters in the *Administration Console Help*.

- A client can call `WLProducer.setUnitOfOrder(name)` and change the initial connection factory setting on the producer.

- Configure a standalone or distributed destination to always use a system-generated Unit-of-Order name. See the following topics in the *Administration Console Help*:

    – Configure advanced topic parameters

    – Configure advanced queue parameters

    – Uniform distributed topics - configure advanced parameters

    – Uniform distributed queues - configure advanced parameters

    – Configure advanced JMS template parameters

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

Unit-of-Order Naming Rules

A Unit-of-Order is identified by a name attribute. Within a destination, messages that have the same value for the Unit-of-Order name attribute belong to the same Unit-of-Order. The name can be provided by either the system or the application. Messages in the same Unit-of-Order all share the same name. See "How to Create a Unit-of-Order" on page 10-8.

The name attribute for a Unit-of-Order must adhere to the following rules:

- A valid value for the Unit-of-Order name attribute is any non-null and non-empty string.

- System-generated Unit-of-Order names are timestamp-based and statistically unique.

- Applications can supply their own Unit-of-Order names. For example, WebLogic Integration applications can use Workflow names and Web Services applications can use conversation names.

- Message Unit-of-Order has its own name space. A Unit-of-Order does not need to be unique with respect to other named objects. For instance, it is valid to have a Unit-of-Order named *Foo* and a queue named *Foo*.

- The scope of a Message Unit-of-Order is limited to a single destination. Two different Units of Order on two destinations can have the same name.

- One or more producers can send messages with the same Unit-of-Order name by using the same string to create the Unit-of-Order. The Unit-of-Order name can be extracted from a delivered message. Example:

  ```
  msg.getStringProperty("JMS_BEA_UnitOfOrder");
  ```

  So a system-generated Unit-of-Order name can be used by more than one producer. This paradigm works just as well for application-assigned Unit-of-Order names. It will be most efficient if the information is serialized in only one place, so a property like Conversation ID can be stored only as the Unit-of-Order name. This paradigm does not work when the message has been sent through a non-Unit-of-Order JMS provider (releases prior to WebLogic 9.0 or non-WebLogic JMS providers).

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 10-11

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.

1. Messages D through Z are undeliverable until messages A, B, and C are acknowledged.

2. Messages A, B, and C are acknowledged or recovered.

3. Message D is available to the message delivery system.

4. Message D does not pass the filter and can never be presented to Consumer1.

5. Messages E through Z are undeliverable until message D is processed.

- The transaction that contains message D must be rolled back.

- Once message D is processed, messages E through Z can be delivered.

For more information, see "Filtering Messages" on page 5-34.

What Happens When Destination Sort Keys are Used

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

For example, messages A and B arrive and in the same Unit-of-Order on a queue that is sorted by priority and the sort order is depending, but message B has a higher priority than A.

Even though message B has a higher priority than message A, message B is still not deliverable until message A 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 Destinations

As previously discussed in the "Message Processing According to the JMS Specification" on page 10-2, the Java Message Service Specification does not guarantee ordered message delivery when applications use distributed queues. WebLogic JMS directs messages with the same Unit-of-Order and having a distributed destination target to the same distributed destination member. The member is selected by the destination's Unit-of-Order configuration:

- "Using the Path Service" on page 10-12

- "Using Hash-based Routing" on page 10-13

Using the Path Service

You can configure the WebLogic Path Service to provide a persistent map that can store the information required to route the messages contained in a Unit-of-Order to its destination resource—a member of a uniform distributed destination. If the WebLogic Path Service is configured for a uniform distributed destination, the routing path to a member destination 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*.

Using Hash-based Routing

If the WebLogic Path Service is not configured, the default routing path to a uniform queue member is chosen by the server based on the hash codes of the Message Unit-of-Order name and the uniform distributed queue members. An advantage of this routing mechanism is that routes to a distributed queue member are calculated quickly and do not require persistent storage in a cluster.

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

- If a distributed queue member has an associated Unit-of-Order and is removed from the distributed queue, new messages are sent to a different distributed queue member and the messages will not be continuous with older messages.

- If a distributed Queue member has an associated Unit-of-Order and is unreachable, the producer sending the message will throw a `JMSOrderException` and the messages are not routed to other distributed Queue members. The exception is thrown because the JMS messaging system can not meet the quality-of-service required — only one distributed destination member consumes messages for a particular Unit-of-Order.

Configuring Routing on Uniform Distributed Destinations

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

- Uniform distributed topics - configure advanced parameters in the *Administration Console Help*

- Uniform distributed queues - configure advanced parameters in the *Administration Console Help*

Using Unit-of-Order with Topics

Assigning a Unit-of-Order does not prohibit parallel processing of a message by two subscribers on the same topic. Since individual subscribers for a topic have their own destination and message list, similar to a queue with one consumer, messages are processed by all subscribers according to the Unit-of-Order assigned at the time of production.

If messages are sent to a distributed topic, the order of the messages on a particular physical member is defined by the order the messages arrive at the member. See "Publish/Subscribe Messaging" on page 2-5.

Using Unit-of-Order with JMS Message Management

JMS message management allows a JMS administrator to move and delete most messages in a running JMS Server. This allows an administrator to violate the delivery rules specified in "Message Delivery with Unit-of-Order" on page 10-3.

If messages A, B, C, and D are produced and sent to destination D1 and belong to Unit-of-Order *foo*, consider the following:

- Moving messages C and D to destination D2 may allow parallel processing of messages from both destinations.

- Moving messages B and C to destination D2 may allow parallel processing of message A and messages B and C. After message A is processed, message D is deliverable.

For applications that depend on maintaining message order, a best practice is to move all of the messages in a Unit-of-Order as a single group.

To ensure Unit-of-Order delivery rules are maintained, use the following steps:

1. Pause the source destination and the target destination.

2. Select all of the messages with the Unit-of-Order you would like to move.

3. Move the selected messages to the target destination. If necessary, sort them according to the order that you want them processed.

4. Resume the source and target destinations.

For more information, see Troubleshooting WebLogic JMS in *Configuring and Managing WebLogic JMS*.

Using Unit-of-Order with WebLogic Store-and-Forward

WebLogic Store-and-Forward supports Message Unit-of-Order. For example, a Store-and-Forward producer sends messages with a Unit-of-Order named *Foo*. If the producer disconnects and reconnects through a different connection, the producer creates another Unit-of-Order with the name *Foo* and continues sending messages. All messages sent before and after the reconnect are directed through the same Store-and-Forward agent. See *Configuring and Managing WebLogic Store-and-Forward*.

Using Unit-of-Order with WebLogic Messaging Bridge

If both the source and target destinations are WebLogic Server 9.0 or later Messaging Bridge instances, you can enable `PreserveMsgProperty` on the Messaging Bridge to preserve the Unit-of Order name and set the producer's Unit-of-Order accordingly. See *Configuring and Managing WebLogic Messaging Bridge*.

Limitations of 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-of-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).

Using Unit-of-Work Message Groups

The following sections describe how to use Unit-of-Work Message Groups to provide groups of messages when using WebLogic JMS:

What Are Unit-of-Work Message Groups?

Many applications need an even more restricted notion of a group than provided by the Message Unit-of-Order (UOO) feature. Therefore, release 9.2 introduces the Unit-of-Work (UOW) Message Groups, which allows applications to send JMS messages, identifying some of them as a group and allowing a JMS consumer to process them as such. For example, an JMS producer can designate a set of messages that need to be delivered to a single client without interruption, so that the messages can be processed as a unit. Further, the client will not be blocked waiting for the completion of one unit when there is another unit that is already complete.

Caution: It is a programming error to use both the Unit-of-Order and Unit-of-Work features on the same JMS message.

The following sections describe how to use Message Unit-of-Work to provide strict message grouping when using WebLogic JMS:

- "Understanding Message Processing With Unit-of-Work" on page 11-2
- "How to Create a Unit-of-Work Message Group" on page 11-6
- "Message Unit-of-Work Advanced Topics" on page 11-13
- "Limitations of UOW Message Groups" on page 11-15

Understanding Message Processing With Unit-of-Work

These sections provide basic conceptual information about UOW message groups.

Basic UOW Terminology

Table 11-1 defines the terms used to define UOW.

Table 11-1 Unit-of-Work Terminology

Term	Definition
Unit of Work (UOW)	A set of JMS messages that need to be processed as a single unit.
UOW Component Message	A message that is part of a UOW. In order for WebLogic JMS to identify a message as part of a UOW, the message must have the JMS properties described in "How To Write a Producer to Set UOW Message Properties" on page 11-6.

Table 11-1 Unit-of-Work Terminology (Continued)

Term	Definition
UOW Producer	A producer that needs to split its work into multiple parts (i.e., a creator of a UOW). Multiple producers can concurrently contribute component messages to a UOW message, as illustrated in "Message Unit-of-Work Case Study" on page 11-4.
	If fact, a UOW producer can close midway through a UOW and a new producer can complete the UOW message, while maintaining the same strict component message integrity (e.g., detect duplicates, etc.).
Intermediate Destination	A destination whose consumers have the job of processing component messages separately rather than as a unit. No special UOW configuration is required for intermediate destinations.
	When a component message arrives on an intermediate destination it will be made available without waiting for other component messages to arrive. Further, if the intermediate destination is a distributed destination, no special routing need occur. See "How to Write a UOW Consumer and/or Producer For an Intermediate Destination" on page 11-9.
Terminal Destination	A destination whose consumers have the job of processing a full UOW. A destination is identified as a *terminal destination* by the Unit-of-Work Message Handling Policy parameter on standalone destinations, distributed destinations, or JMS templates. See "Configuring Terminal Destinations" on page 11-10.
Available/Visible Messages	Equivalent JMS terms that refer to a message becoming ready for consumption, pending the reception of any messages that precede it. For example, a JMS message is not available until its birth time has been reached or a JMS message that is sent as part of a transaction is not visible until that transaction is committed.

Rules For Processing UOW Messages

The following rules apply to UOW messages.

Rule One: All Messages Required For Processing

No message within the UOW will be available until all of them are available on the terminal destination.

Rule Two: Message Reordering
> No matter what order the messages arrive to the terminal destination, they will be put into the order specified by the UOW producer.

Rule Three: Gap Freedom
> The group of messages will be delivered to the user without gaps. That is, all messages in the group will be delivered to the user before messages from any other group (or part of no group at all).

Rule Four: Single Consumer Consumption
> The group of messages will be delivered to the same consumer.

Message Unit-of-Work Case Study

This section provides a simple case study for Message Unit-of-Work based on an online order that requires a variety of merchandise from multiple companies.

Jill Orders Miscellaneous Items From an Online Retailer

The *Megazon* online retailer implements a processing design that uses JMS to process customer orders for a variety of merchandise, some of which need to be routed to Megazon's partner companies to complete the order. For example, Megazon can directly fulfill book orders, but must re-route some parts of the order for certain electronics or houseware items. Since Megazon is configured to use UOW, items in an order can be routed as UOW messages to these intermediate company destinations before being passed onto Megazon's terminal destination where all the UOW messages that comprise the order are gathered before a final invoice can be processed.

The Megazon JMS processing system is composed of:

- A UOW producer sending order fulfillment component messages with the required UOW properties to the appropriate intermediate and/or terminal destinations.

- Intermediate destinations for non-book items, where UOW component messages are processed by consumer and/or producer clients before being passed onto the final UOW destination.

- A UOW terminal destination where the component messages are gathered for final processing.

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

How Message Unit-of-Work Completes the Order

To ensure that all messages in Jill's order are processed as a single unit, the order-taking JMS producer client sets UOW properties on her order messages to indicate that they are part of a single unit. These UOW message properties must also be copied by any consumer and/or producer clients listening on the intermediate Gadget Planet, Widget World, and Desperate Housewares destinations before they pass the UOW messages onto the terminal destination. Last, the system administrator for Megazon configures the terminal destination to UOW Message Handling Policy parameter to Single Message Delivery. See "How to Create a Unit-of-Work Message Group" on page 11-6.

The following diagram and corresponding actions demonstrate how Jill's order was processed using Message Unit-of-Work.

Figure 11-1   Workflow for Jill's Order Using Unit-of-Work
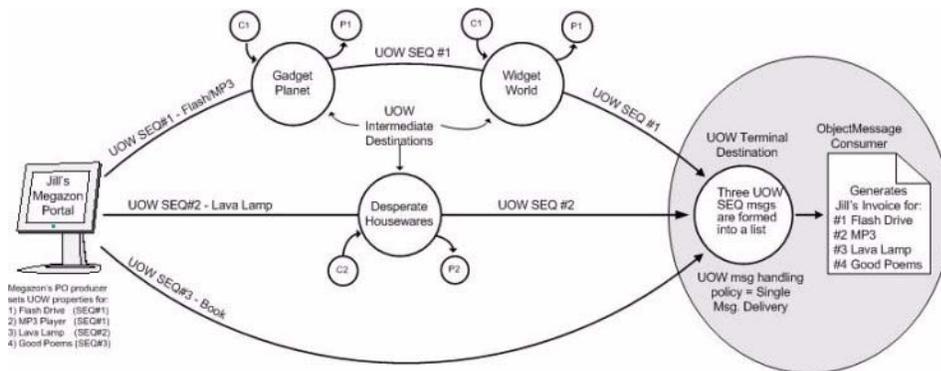
1. Jill clicks the order button from her shopping cart.

2. The order is split into three messages that use the same unique UOW name:

    – SEQ#1, which is routed to the intermediate Gadget Planet queue, where a consumer processes the Flash Drive order before passing SEQ#1 onto a producer who then routes it to the intermediate Widget World queue, where a consumer processes the MP3 player order before passing SEQ#1 to the terminal Megazon queue for final invoice processing.

    – SEQ#2, which is routed to the intermediate Desperate Housewares queue, where a consumer processes the lava lamp order before passing SEQ#1 onto a producer who routes it to the Megazon terminal processing queue for final invoice processing.

– SEQ#3, which is routed directly to Megazon's terminal queue for book order fulfillment and for final invoice processing.

3. The terminal Megazon queue gathers the three UOW messages before forming them into an `ObjectMessage` list for delivery to Megazon's invoice consumer client.

4. Jill receives an invoice that shows her entire order has been processed.

How to Create a Unit-of-Work Message Group

The following sections describe how to create a Message Unit-of-Work.

- "How To Write a Producer to Set UOW Message Properties" on page 11-6
- "How to Write a UOW Consumer and/or Producer For an Intermediate Destination" on page 11-9
- "Configuring Terminal Destinations" on page 11-10
- "How to Write a UOW Consumer For a Terminal Destination" on page 11-11

How To Write a Producer to Set UOW Message Properties

UOW enables a producer to split its work into multiple parts to accomplish its goal. UOW is, in effect, taking these multiple messages and joining them into one. Whether component messages are delivered as parts of a single message or as many messages, it is easiest to envision them as a single virtual message, as well as individual messages.

In order for WebLogic JMS to identify a message as part of a UOW, the message must have the JMS properties in Table 11-2 set by the producer client.

Table 11-2 Unit-of-Work Properties

Type	Description
`JMS_BEA_UnitOfWork`	A string property that is set by the standard JMS mechanism for setting properties. For example:
	`message.setStringProperty("JMS_BEA_UnitOfWork", "MyUnitOfWorkName")`
	To avoid naming conflicts, the UOW ID should never be reused. For example, if messages are lost or retransmitted, then they may be perceived as part of a separate UOW. For this reason, BEA recommends using a Java universally unique identifier (UUID). See http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html.
`JMS_BEA_UnitOfWorkSequenceNumber`	An integer property that is set by the standard JMS mechanism for setting properties. For example,
	`message.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber", 5)`
	The legal values are integers greater than or equal to 1
`JMS_BEA_IsUnitOfWorkEnd`	A boolean property that is set by the standard JMS mechanism for setting properties. For example:
	`message.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd", true)`
	When this property is true, the message is the last in the unit-of-work. When this property is false or nonexistent, the message is not last in the unit-of-work.

If the `UnitOfWork` property is not set, then `SequenceNumber` and `End` will be ignored.

Example UOW Producer Code

The following sample client code sample sets the UOW properties defined in Table 11-2.

Listing 11-1   Sample UOW Producer Message Properties

```
for (int i=1; i<=100; i++)
{
```

```
sendMsg.setStringProperty("JMS_BEA_UnitOfWork","joule");
sendMsg.setIntProperty("JMS_BEA_UnitOfWorkSequenceNumber",i);
if (i == 100)
{
  System.out.println("set the end of message flag for message # " + i);
   sendMsg.setBooleanProperty("JMS_BEA_IsUnitOfWorkEnd",true);
}
qSender.send(sendMsg, DeliveryMode.PERSISTENT,7,0);
}
```

UOW Exceptions

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

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

- `BadSequenceNumberException` – This will occur if (a) `UnitOfWork` is set on the message, but `SequenceNumber` is not or (b) the `SequenceNumber` is less than or equal to zero.

- `OutOfSequenceRangeException` – This will be thrown if (a) a message is sent with a SequenceNumber that is higher than the sequence number of the message which has already been marked as the end of the unit or (b) a message is sent with a sequence number which is lower than a message which has already arrived in the same unit, yet the new message is marked as the end message.

- `DuplicateSequenceNumberException` – This will be thrown to the producer if it sends a message with a sequence number which is the same as a previously sent message in the same UOW.

- `JMSException` – A JMS exception will be thrown if a message has both the `UnitOfOrder` property set and the `UnitOfWork` property set.

Tip: As a programming best-practice, consider having your UOW producers send all component messages that comprise a new UOW under a single transaction. This way, either all of the work is completed or none of it is. For example, if a UOW producer gets an exception or crashes partway through a UOW and wants to then cancel the current

UOW, then the entire transaction will be rolled back and the application will not need to make a decision for each message after a failure.

How to Write a UOW Consumer and/or Producer For an Intermediate Destination

An *intermediate destination* is one whose consumers have the job of processing component messages separately rather than as a unit. A JMS ForwardHelper extension API is available to assist developers who are writing producers and/or consumers at intermediate destinations. This is because there are many message properties that need to be copied from the incoming message to the outgoing message. For example, the message properties that control the behavior of UOW need to be copied.

The following intermediate consumer code sample copies the UOW properties defined in Table 11-2.

Listing 11-2 Sample Client Code for UOW Intermediate Destination

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

Note that the three UOW properties are copied from the incoming message to the outgoing message.

Configuring Terminal Destinations

A destination is identified as a *terminal destination* by the Unit-of-Work Message Handling Policy parameter on standalone destinations, distributed destinations, or JMS templates. There is also a parameter that allows for expiration of incomplete work on terminal destinations.

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

Table 11-3 Unit-of-Work Configuration Options

Console Label/MBean Name	Description
Unit-of-Work (UOW) Message Handling Policy `UnitOfWorkHandlingPolicy`	Specifies whether the Unit-of-Work (UOW) feature is enabled for a destination. • `Pass-Through` – By default, destinations do not treat messages as part of a UOW. • `Single Message Delivery` – Select this option if UOW consumers are receiving component messages on this terminal destination. When selected, component UOW messages are formed into a list and are consumed as an `ObjectMessage` containing the `java.util.list`.
Expiration Time for Incomplete UOW Messages `IncompleteWorkExpirationTime`	The maximum length of time, in milliseconds, before undeliverable messages in an incomplete UOW are expired. Such messages will then follow the expiration policy defined for undeliverable messages. Message expiration begins once the first UOW message arrives. This field is effective only if Unit-of-Work Handling Policy is set to `Single Message Delivery`. The default value of -1 means that UOW messages will never expire. **Note:** If an expiration time is not configured on terminal destination, it is possible for a UOW message to wait indefinitely on the destination when a component message was either: (A) never sent/committed, (B) expired, or (C) manually deleted.

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

- Configure advanced topic parameters

- Configure advanced queue parameters

- Uniform distributed topics - configure advanced parameters

- Uniform distributed queues - configure advanced parameters

- Configure advanced JMS template parameters

For more information about these parameters, see `DestinationBean` and `TemplateBean` in the *WebLogic Server MBean Reference.*

UOW Message Routing for Terminal Distributed Destinations

The Unit-of-Order Routing field is used to determine the routing of UOW messages for uniform distributed destinations, using either the Path Service or Hash-based Routing. And similar to UOO, when a UOW terminal destination is also a distributed destination, all messages within a UOW must go to the same distributed destination member. For more information on the UOO routing mechanisms, see Using Unit-of-Order With Distributed Destinations in *Programming WebLogic JMS.*

However, basic UOO routing and UOW routing are not exactly the same. Strictly, all messages within a single UOO do not have to go to the same member: when there are no more unconsumed messages for a certain UOO, newly arrived messages can go to any member. In UOW, message routing must be guaranteed until the *whole* UOW has arrived at the physical destination and consumption is irrelevant.

How to Write a UOW Consumer For a Terminal Destination

The following sample UOW consumer code shows how a consumer listening on a terminal destination verifies that all component messages sent are contained within the final UOW message.

Listing 11-3   Sample Client Code For UOW Terminal Destination

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

Message Unit-of-Work Advanced Topics

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

Message Property Handling

UOW is, in effect, taking multiple messages and joining them into one. This is true whether the messages are delivered as one message or not. For example, each message will have an independent expiration time, but if one expires, none of them will ever be delivered. Therefore, as a best practice your message producers should make sure that messages that make up a UOW are as uniform as possible.

Whether component messages are delivered as parts of a single message or as many messages, it is easiest to envision them as a single *virtual* message, as well as individual messages. For example, since the messages need to be seen consecutively, UOW's effect on message sorting can be viewed as determining the correct placement of the virtual message. The same is true of message selection (a consumer must see the whole group or not see the group at all); WebLogic JMS must determine whether "*consumer A* must see the virtual message" before deciding to deliver all of the messages to *consumer A*.

System-Generated Properties

Some fields of the virtual message will need to be populated independent of the component messages. For example, the virtual message cannot get its value for delivery count from a component message. This is the complete list of property values that are system-generated:

- Timestamp

- Delivery count (redelivered)

- Destination

Final Component Message Properties

Otherwise, the message properties will be derived from the component messages. However, different properties get values derived in different ways, as suits their nature. One way to derive virtual message properties is to get their values directly from one of the component messages (this simplifies the handling of component messages with different property values). For simplicity, the last message in the UOW is the message from which the values are derived. For example, the

message priority for the virtual message will be the priority of the message marked as last (by having the property JMS_BEA_IsUnitOfWorkEnd set to true).

This is the complete list of virtual message properties that are derived from the values contained in the last message in the UOW:

- Message ID
- Correlation ID
- Priority
- User Properties
- User ID

Component Message Heterogeneity

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

- Delivery Mode
- Expiration

ReplyTo Message Property

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

UOW and Uniform Distributed Destinations

As discussed in "UOW Message Routing for Terminal Distributed Destinations" on page 11-11, the Unit-of-Order Routing field is used to determine the routing mechanism for UOW messages. One other requirement for UOW in distributed destinations is that all member destinations must have the same value for the UOW Handling Policy. A configuration that is configured otherwise is invalid.

As a best practice, the use of topics (especially distributed topics) is discouraged for use as intermediate UOW destinations, as this configuration may possibly lead to duplicate component messages.

UOW and Store-and-Forward Destinations

The WebLogic Store-and-Forward service supports UOW, with the exception that a store-and-forward (SAF) imported destination cannot be a terminal destination. However, SAF obeys the routing rules of UOW messages, just as it does for UOO messages. See Using Unit-of-Order With WebLogic Store-and-Forward in *Programming WebLogic JMS*.

Limitations of UOW Message Groups

This section provides additional general information to consider when using UOW.

- JMS clients created using Weblogic Server 8.1 or earlier cannot create messages that will be processed as part of a UOW.

- The JMS C JNI client is not able to process UOW messages at a terminal destination, since they are object messages. It can, however, be used as a UOW producer or on an intermediate destination.

- UOW is poorly suited for sets of large file transfers. Ideally, your messaging environment is configured for lower max message sizes and to facilitate the streaming transfer of large chunks of data (such as large files) from a single producer to a single consumer. UOW doesn't handle this use-case because the individual messages are accumulated back into one giant message on the server before they are pushed to the consumer, rather than streamed.

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 12-4.

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 5-15.

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 12-1   Setting Up and Using a JMS Transacted Session
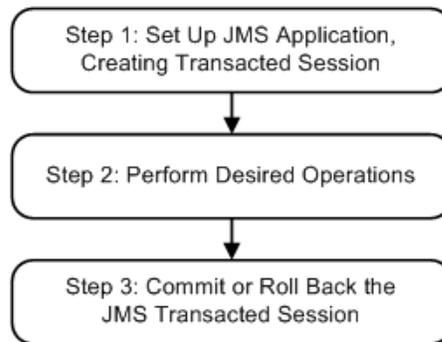
Step 1: Set Up JMS Application, Creating Transacted Session

Set up the JMS application as described in "Setting Up a JMS Application" on page 4-2, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-6, 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.

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 12-2   Setting Up and Using a JTA User Transaction
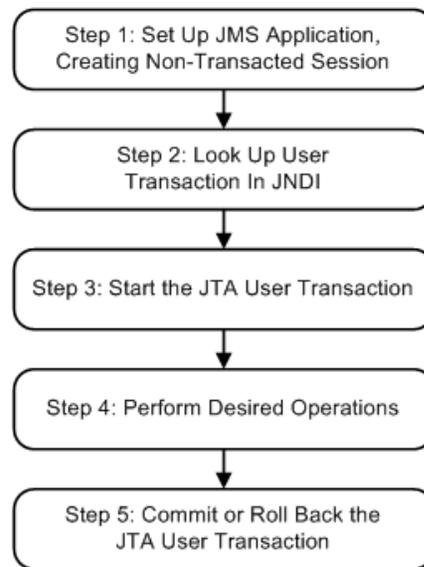
Step 1: Set Up JMS Application, Creating Non-Transacted Session

Set up the JMS application as described in "Setting Up a JMS Application" on page 4-2, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on

page 4-6, 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 EJBs" 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 4-2.

```
//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()
```

WebLogic JMS C API

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

What Is the WebLogic JMS C API?

The WebLogic JMS C API is an application program interface that enables you to create C client applications that can access WebLogic JMS applications and resources. The C client application then uses the Java Native Interface (JNI) to access the client-side Java JMS classes. See Figure 13-1.

For this release, the WebLogic JMS C API adheres to the JMS Version 1.1 specification to promote the porting of Java JMS 1.1 code. For more information, see the WebLogic JMS C API Javadocs.

Figure 13-1 WebLogic JMS C API Client Application Environment
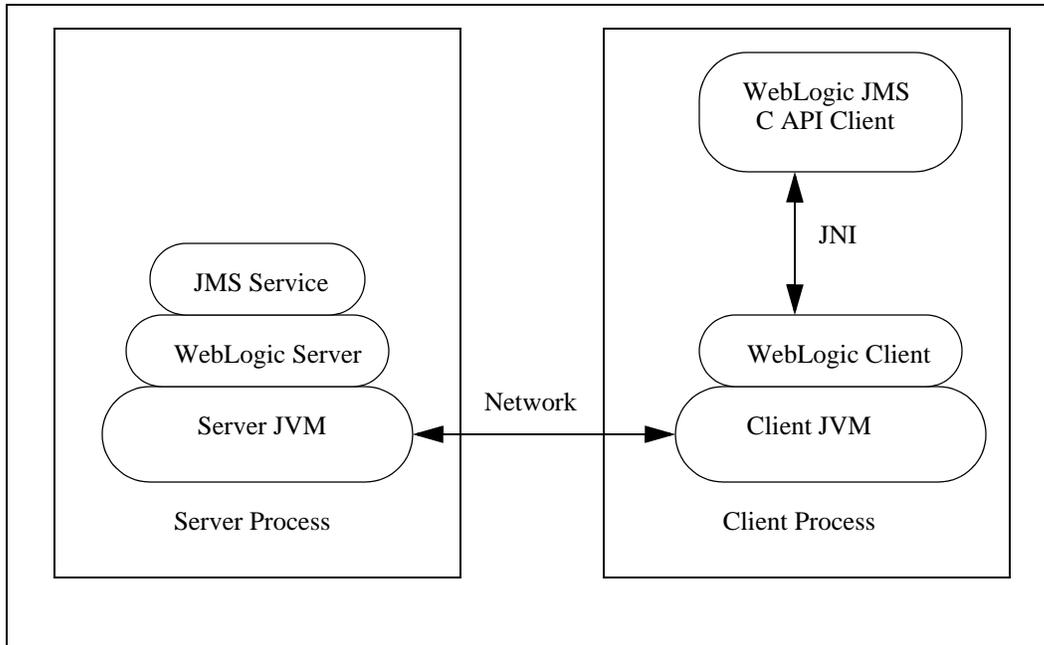
System Requirements

The following section provides information on the system requirements needed to use the WebLogic JMS C API in your environment:

- A list of supported operating systems for the WebLogic JMS C API is located at Supported Interoperability Tools in *Supported Configurations for WebLogic Platform.*

- A supported JVM for your operating system. See WebLogic Platform 9.0 Supported Configurations in *Supported Configurations for WebLogic Platform.*

- An ANSI C compiler for your operating system.

- Use one of the following WebLogic clients to connect your C client applications to your JMS applications:

  – The WebLogic application client (`weblogic.jar` file).

– The WebLogic JMS thin client (`wljmsclient.jar` file). See the WebLogic JMS Thin Client in *Programming Stand-alone Clients*.

WebLogic JMS C API Code Examples

BEA Systems provides samples for JMS developers that illustrate how to configure and develop the WebLogic JMS C API clients. See http://dev2dev.bea.com/code/index.jsp.

Design Principles

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

- "Java Objects Map to Handles" on page 13-3

- "Thread Utilization" on page 13-3

- "Exception Handling" on page 13-4

- "Type Conversions" on page 13-4

- "Memory Allocation and Garbage Collection" on page 13-6

- "Closing Connections" on page 13-6

- "Helper Functions" on page 13-6

Java Objects Map to Handles

The WebLogic JMS C API is handle-based to promote modular code implementation. This means that in your application, you implement Java objects as handles in C code. The details of how a JMS object is implemented is hidden inside a handle. However, unlike in Java, when you are done with a handle, you must explicitly free it by calling the corresponding `Close` or `Destroy` methods. See "Memory Allocation and Garbage Collection" on page 13-6.

Thread Utilization

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

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

- `java.jms.Connection` objects are thread safe, and the corresponding WebLogic JMS C API handle, `JmsConnection`, is thread safe.

As long as concurrency control is managed by the C client application, all objects returned by the WebLogic JMS C API 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` represents the class `java.lang.Throwable`.

- `JavaException` represents the class `java.lang.Exception`.

- `JmsException` represents the class `javax.jms.JMSException`. All standard subclasses of `JMSException` are determined by bits in the type descriptor of the exception. The type descriptor is returned with a call to `JmsGetLastException`.

Type Conversions

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

Integer (int)

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

Long (long)

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

Character (char)

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

String

`String` converts to `JmsString`.

Java strings are arrays of 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)`

- `JavaString (UNISTRING)`

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

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

The `UNISTRING` encoding encodes every 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 the `stringType` element of `JmsString` is used as output, the caller has the option to let the API allocate enough space for output using `malloc`, or you can supply the space and have the system copy the returned string into the provided bytes. If the appropriate field in the union (either string or data) is NULL, then the API allocates enough space for the output using `malloc`. It is the callers responsibility to free this allocated space using `free` when the memory is no longer in use. If the appropriate field in the union (string or data) is not NULL, then the `allocatedSize` field of `JmsString` must contain the number of bytes available to be written.

If there is not enough space in the string to contain the entire output, then `allocatedSize` sets to the amount of space needed and the API called returns `JMS_NEED_SPACE`. The appropriate field in the `JmsString` (either string or data) contains as much data as could be stored up to the `allocatedSize` bytes. In this case, the NULL character may or may not have been written at the end of the C string data returned. Example:

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 you are able to extract the string from the message handle. Alternatively, you can free the original buffer and allocate a new buffer of the correct size.

Memory Allocation and Garbage Collection

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

Implementation Guidelines

Be aware of the following when you implement the WebLogic JMS C API:

- It does not support WebLogic Server JMS extensions, including XML messages.

- It does not support JMS Object messages.

- It creates an error log if an error is detected in the client. This error log is named `ULOG.mmddyy` (month/day/year). This log file is fully internationalized using the `NLSPATH`, `LOCALE`, and `LANG` environment variables of the client.

- Users who want to translate the message catalog can use the `gencat` utility provided on Windows or the `gencat` utility of the host platform. If the generated catalog file is placed according to the `NLSPATH`, `LOCALE`, and `LANG` variables, then the translated catalog will be used when writing messages to the log file.

- You can set the following environment variables in the client environment:

    - `JMSDEBUG`— Provides verbose debugging output from the client.

    - `JMSJVMOPTS`—Provides extra arguments to the JVM loaded by the client.

    - `ULOGPFX`— Configures the pathname and file prefix where the error log file is placed.

WebLogic JMS C API

Recovering from a Server Failure

These sections discuss how WebLogic JMS client applications reconnect or recover from a server/network failure, and how to migrate JMS data after a server failure.

- "Automatic JMS Client Failover" on page 14-2

- "Programming Considerations for WebLogic Server 9.0 or Earlier Failures" on page 14-16

- "Migrating JMS Data to a New Server" on page 14-16

Automatic JMS Client Failover

With the automatic JMS client reconnect feature, if a server or network failure occurs, some JMS client objects will transparently failover to use a another server instance, as long as one is available. For example, if a fatal server failure occurs, JMS clients automatically attempt to reconnect to the server when it becomes available.

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

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

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

For more information, refer to the following sections:

- "Automatic Failover for JMS Producers" on page 14-3

- "Configuring Automatic Failover for JMS Consumers" on page 14-7

- "Explicitly Disabling Automatic Failover on JMS Clients" on page 14-13

- "Limitations for Automatic JMS Client Failover" on page 14-14

- "Best Practices for JMS Clients Using Automatic Failover" on page 14-14

Automatic Failover for JMS Producers

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

- Connection

- Session

- MessageProducer

If you do not want your JMS clients to be automatically reconnected, then you must explicitly disable this feature either programatically or administratively, as described in "Explicitly Disabling Automatic Failover on JMS Clients" on page 14-13.

Sample Producer Code

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

Listing 14-1   Sample JMS Client Code for Message Production

```
0.    Context ctx = create WebLogic JNDI context with credentials etc.
1.    ConnectionFactory cf = ctx.lookup(JNDI name of connection factory)
2.    Destination dest    = ctx.lookup(JNDI name of destination)
      // the following operations recover from network failures
3.    Connection  con     = cf.createConnection()
4.    Session     sess    = con.createSession(no transactions, ack mode)
5.    MessageProducer prod = sess.createProducer(dest)
6.    Loop over:
7.       Message msg = sess.createMessage()
8.       prod.send(msg)
9.    con.close(); ctx.close()
```

The JMS producer will transparently failover to another server instance, if one is available. This keeps the client code as simple as listed above and eliminates the need for client code for retrying across network failures.

The WebLogic JMS does not reconnect MessageConsumers by default. For this to automatically occur programmatically, your client application code must call the WebLogic `WLConnection` extension, with the `setReconnectPolicy` set to "all", as explained in "Configuring Automatic Failover for JMS Consumers" on page 14-7.

Re-usable ConnectionFactory Objects

Since Weblogic Server 8.1, a ConnectionFactory object looked up via JNDI (see Step 1 in Listing 14-1 and Listing 14-2) is re-usable after a server or network failure without requiring a re-lookup. A network failure could be between the JMS client JVM and the remote WebLogic Server instance it is connected to as part of the JNDI lookup, or between the JMS client JVM and any remote WebLogic Server instance in the same cluster where the JMS client subsequently connects.

Re-usable Destination Objects

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

A network failure could be between the client JVM and the WebLogic Server instance it is connected to, or between that WebLogic Server instance and the WebLogic Server instance that actually hosts the destination. The Destination object will also be robust after restarting the WebLogic Server instance hosting the destination.

Note: For information on how consumers of distributed destinations behave with automatic JMS client reconnect, see "Consumers of Distributed Destinations" on page 14-11.

Reconnected Connection Objects

The JMS Connection object is used to map one-to-one to a physical network connection between the client JVM and a remote WebLogic Server instance. With the JMS client reconnect feature, the JMS Connection object that the client gets from the `ConnectionFactory.createConnection()` method (see Step 3 in Listing 14-1 and Listing 14-2) maps in a one-to-one-at-a-time fashion to the physical network connection. One

consequence is that while the JMS client continues to use the same Connection object, it could be actually communicating with a different WebLogic Server instance after an implicit failover.

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

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

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

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

Special Cases for Reconnected Connections

There are special cases that can occur when producer connections are refreshed:

- *Connections with a ClientID for Durable Subscribers* – If your Reconnect Policy field is set to **None** or **Producer**, and a JMS Connection has a Client ID specified at the time of a network/server failure, then the Connection will not be automatically refreshed. The reason for this restriction is backward compatibility, which avoids breaking existing JMS applications that try to re-create a JMS Connection with the same connection name after a failure. If implicit failover also occurs on a network failure, then the application's creation of the connection will fail due to a duplicate ClientID.

  Note: For information on how a consumer connection with a ClientID behaves, see .

- *Closed Objects Are Not Refreshed* – When the application calls `javax.jms.Connection.close()`, `javax.jms.Session.close()`, etc., that object and

it descendents are not refreshed. Similarly, when the JMS client is told its Connection has been administratively destroyed, it is not refreshed.

- *Connection with Registered Exception Listener* – If the JMS Connection has an application ExceptionListener registered on it, that ExceptionListener's `onException()` callback will be invoked even if the connection is implicitly refreshed. This notifies the application code of the network disconnect event. The JMS client application code might normally call `connection.close()` in `onException`; however, if it wants to take advantage of the reconnect feature, it may choose not to call `connection.close()`. The registered ExceptionListener is also migrated transparently to the internally refreshed connection to listen for exceptions on the refreshed connection.

- *Multiple Connections* – If there are multiple JMS Connections created off the same ConnectionFactory object, each connection will behave independently of the other connections as far as the reconnect feature is concerned. Each connection will have its own connection status, its own connection retry machinery, etc.

Reconnected Session Objects

As described in "Reconnected Connection Objects" on page 14-4, JMS Session objects are refreshed when their associated JMS connection gets refreshed (see Step 4 in Listing 14-1 and Listing 14-2). Session states, such as acknowledge mode and transaction mode, are preserved across each refresh occurrence. The same session object can be used for calls, like `createMessageProducer()`, after a refresh.

Special Cases for Reconnected Sessions

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

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

  After a Session refresh, operations like `send()` or `receive()` will not throw an exception; it is only the first commit after a refresh that will throw an exception. However, the first commit after a Session refresh will not throw an exception if there were no pending transactional operations in that JMS session at the time of the network disconnect. In case of `Session.commit()` throwing the exception, the client application code can simply retry

all the operations in the transaction again with the same (implicitly refreshed) JMS objects. The stale operations before a refresh will not be committed and will not be duplicated.

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

Reconnected MessageProducer Objects

As described in "Reconnected Connection Objects" on page 14-4, JMS MessageProducer objects are refreshed when their associated JMS connection gets refreshed (see Step 5 in Listing 14-1). If producers are non-anonymous, that is, they are specific to a Destination object (standalone or distributed destination), then the producer's destination is also implicitly refreshed, as described in "Re-usable Destination Objects" on page 14-4. If a producer is anonymous, that is not specific to a Destination object, then the possibly-stale Destination object specified on the producer's `send()` operation is implicitly refreshed.

Special Case for Reconnected MessageProducers and Distributed Destinations

It is possible that a producer can send a message at the same time that a distributed destination member becomes unavailable. If WebLogic JMS can determine that the distributed destination member is not available, or was not available when the message was sent, the system will retry sending the message to another distributed member. If there is no way to determine if the message made it through the connection all the way to the distributed member before it went down, the system will not attempt to resend the message because doing so may create a duplicate message. In that case, WebLogic JMS will throw an exception. It is up to the application to catch that exception and decide whether or not to resend the message.

Configuring Automatic Failover for JMS Consumers

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

Explicitly enabling automatic refresh of consumers also refreshes Connections with a configured Client ID for a durable subscriber, as described in "Consumer Connections with a ClientID for Durable Subscriptions" on page 14-11. However, refreshed consumers does not include

QueueBrowser clients, which are never refreshed, as described in "Limitations for Automatic JMS Client Failover" on page 14-14.

Sample Consumer Client Code

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

Listing 14-2   Sample JMS Client Code for Message Consumption

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

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

Configuring Automatic Client Refresh Options

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

Table 14-1 Automatic JMS Client Reconnect Options

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

For instructions on configuring client parameters on a connection factory using the Administration Console, see "Configure connection factory client parameters" in the *Administration Console Online Help*. For more information about these parameters, see ClientParamsBean in the *WebLogic Server MBean Reference.*

Common Cases for Reconnected Consumers

This section describes the common scenarios when refreshing synchronous and asynchronous consumers.

Synchronous Consumers

Synchronous consumers use MessageConsumer.receive(), MessageConsumer.receive(timeout), and MessageConsume.receiveNoWait() methods to consume messages. The first two methods are already expected to be potentially block the application code, while the third method is not expected to block the application code. To retain

these semantics, the following rules describe interaction of the reconnect feature with the synchronous consumer calls:

- `MessageConsumer.receive()`– If there is a network disconnect during this call, this method can block for up to Reconnect Blocking Time property (described in the configuration section) for a reconnect to go through before throwing an Exception.

- `MessageConsumer.receive(timeout)` – This call will block for the at-most timeout milliseconds specified by caller. If the Reconnect Blocking Time property is less than timeout, the receive will still block up to the Reconnect Blocking Time setting; if the Reconnect Blocking Time value is more than timeout, the receive will only block up to timeout.

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

If these methods eventually reach their respective timeout/wait periods, they all will throw the same Exceptions. as without reconnect. If a reconnect succeeds while these methods are blocked/called, these methods will continue returning messages, but with a potentially lowered quality-of-service and with generally similar semantics of receiving messages (like Redelivered messages), as after a recover. The application is notified of this possibility by a Connection ExceptionListener callback with `LostServerException`. In addition, for non-`AUTO_ACK` acknowledge modes, the first acknowledge call after a refresh will throw a `LostServerException` to notify the application of this possibility.

Asynchronous Consumers

In the context of a reconnect, the behavior for asynchronous consumers will be governed by the setting on the Total Reconnect Period property. The JMS Consumer's registered message listener's `onMessage()` will continue to be invoked if the reconnect framework is able to successfully re-establish a connection within the Total Reconnect Period setting after a connection failure. If the user explicitly calls a `close()` on the JMS Connection (or on the JMS Session corresponding to the asynchronous Consumer), then the reconnect framework will not invoke any further `onMessages` for that Consumer. The `onMessage()` should expect post-recover behavior (like Redelivered messages) if the Connection ExceptionListener's `onException` is invoked with a `LostServerException`.

Special Cases for Reconnected Consumers

These sections discuss special cases that can occur when consumers are refreshed.

Consumers of Distributed Destinations

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

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

Message-Driven EJBs

Message-driven EJBs (MDBs) are a special sub-case of asynchronous consumers that have their own behavior requirements and their own refresh framework. As such, MDBs are not expected to participate in MessageConsumer refreshes, and are not expected to be affected in any other way by the JMS client reconnect framework.

Consumer Connections with a ClientID for Durable Subscriptions

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

Notes: If a JMS Connection has a ClientID specified at the time of a network/server failure, then reconnecting that client make take significantly longer than your other clients. For example, in a cluster the JMS server must wait for the WebLogic Server "heartbeat" notification that is broadcast from other members of the cluster, as explained in "Failover and Replication in a Cluster" in *Using WebLogic Server Clusters*.

WebLogic JMS does not support durable subscriptions on distributed topics, so there is no issue of failover to another distributed topic member during a refresh.

Non-Durable Subscriptions and Possible Missed Messages

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

Duplicate Messages

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

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

Variations Due to Acknowledge Modes

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

Consumers May Not Reconnect with Migrated JMS Destinations In a Cluster

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

Explicitly Disabling Automatic Failover on JMS Clients

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

Programmatically

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

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

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

Administratively

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

1. Follow the directions for navigating to the JMS Connection Factory: Configuration: Client pages, see Configure connection factory client parameters in the *Administration Console Online Help*.

2. In the Reconnect Policy field, select **None** to disable the JMS client reconnect feature on this connection factory.

    For more information about the Reconnect Policy field, see JMS Connection Factory: Configuration: Client in the *Administration Console Online Help*.

3. Click Save.

For more information about the other JMS connection factory client parameters, see ClientParamsBean in the *WebLogic Server MBean Reference.*

Limitations for Automatic JMS Client Failover

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

- Queue browsers: `javax.jms.QueueBrowser`

- The WebLogic JMS thin client (wljmsclient.jar) will not automatically reconnect. For more information, see "WebLogic JMS Thin Client" in *Programming Stand-alone Clients*.

- Client statistics are reset on each reconnect, which results in the loss historical data for the client.

- Temporary destinations (`javax.jms.TemporaryQueue` and `javax.jms.TemporaryTopic`).

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

Best Practices for JMS Clients Using Automatic Failover

BEA recommends the following best practices for JMS clients when using the Automatic JMS Client Reconnect feature:

Use Transactions to Group Message Work

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

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

JMS Clients Should Always Call the close() Method

As a best practice, your applications should not rely on the JVM's garbage collection to clean up JMS connections because the JMS automatic reconnect feature keeps a reference to the JMS connection. Therefore, always use `connection.close()` to clean up your connections. Also consider using a `Finally` block to ensure that your connection resources are cleaned up. Otherwise, WebLogic Server allocates system resources to keep the connection available.

For more information closing JMS client connections, see "Best Practice: Always Close Failed JMS ClientIDs" on page 5-24.

Programming Considerations for WebLogic Server 9.0 or Earlier Failures

JMS client applications running on Weblogic Server 9.0 or earlier had to reestablish javax.jms objects after a server failure. If you are still running release 9.0 or earlier JMS clients, you may want to program your JMS clients to terminate gracefully in the event of a server failure. For example:

Table 14-2 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 destinations 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 14-3.

Note: There are special considerations when you migrate a service from a server instance that has crashed or is unavailable to the Administration Server. If the Administration Server cannot reach the previously active host of the service at the time you perform the migration, see "Migrating a Service When Currently Active Host is Unavailable" in *Using WebLogic Server Clusters*.

Table 14-3 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.
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	To facilitate recovery after a crash, WebLogic Server provides the Transaction Recovery Service, which automatically attempts to recover transactions on system startup. The Transaction Recovery Service owns the transaction log for a server. For detailed instructions on recovering transactions from a failed server, see "Transaction Recovery After a Server Fails" in *Programming WebLogic JTA*.

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 Avoiding and Recovering From Server Failure in *Managing Server Startup and Shutdown*.

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

Recovering from a Server Failure

APPENDIX A

Deprecated WebLogic JMS Features

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

- "Defining Server Session Pools" on page A-1

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 14-1   Server Session Pool Facility
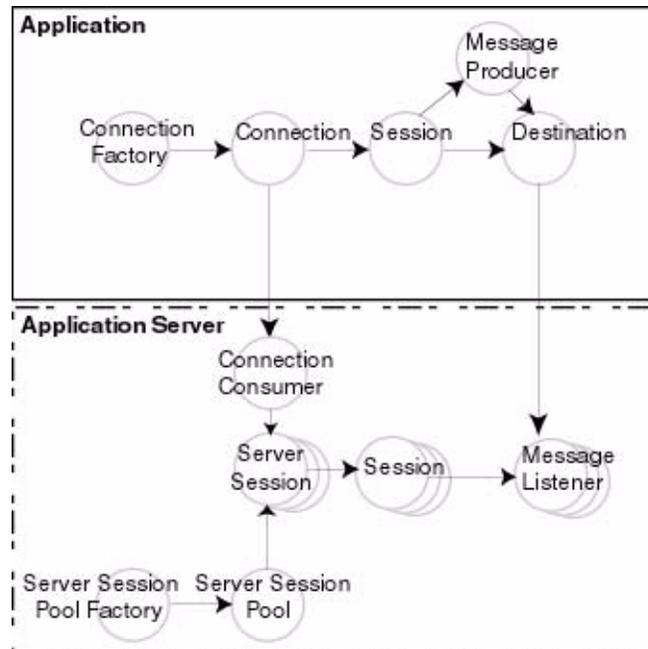
As illustrated in the figure, the application provides a single-threaded message listener. The connection consumer, implemented by JMS on the application server, performs the following tasks to process one or more messages:

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 14-2   Preparing for Concurrent Message Processing
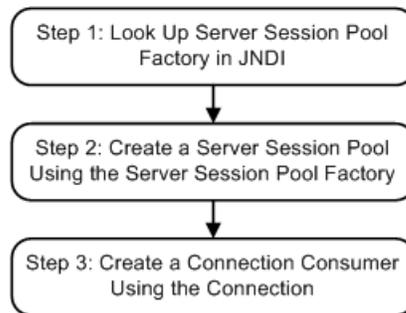
Applications can use other application server providers' session pool implementations within this flow. Server session pools can also be implemented using message-driven beans. For information on using message driven beans to implement server session pools, see "Designing Message-Driven Beans" in *Programming 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" on page 2-27 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" on page 2-28 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" on page 2-28 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 5-34.

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 5-34.

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 5-21.

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 2-9 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 4-17. 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 2-9 or the `javax.jms` Javadoc.

Deprecated WebLogic JMS Features

FAQs: Integrating Remote JMS Providers

The J2EE standards for JMS (messaging), JTA (transaction), and JNDI (naming) work together to provide reliable java-to-java messaging between different host machines and even different vendors. BEA WebLogic Server provides a variety of tools that leverage these APIs to aid integrating remote JMS providers into a local application.

The following sections provide information on how to integrate WebLogic Server with remote JMS providers.

- "Understanding JMS and JNDI Terminology" on page B-2

- "Understanding Transactions" on page B-3

- "How to Integrate with a Remote Provider" on page B-5

- "Best Practices when Integrating with Remote Providers" on page B-7

- "Using Foreign JMS Server Definitions" on page B-9

- "Using EJB/Servlet JMS Resource References" on page B-9

- Using WebLogic Store-and-Forward

- Using WebLogic JMS SAF CLient

- Using a Messaging Bridge

- Using Messaging Beans

- JMS Interoperability Resources

Understanding JMS and JNDI Terminology

Q. What is a remote JMS provider?

A. A remote JMS provider is a JMS server that is hosted outside a local stand-alone WebLogic server or outside WebLogic server cluster. The remote JMS server may be a WebLogic or a non-WebLogic (foreign) JMS server.

Q. What is JNDI?

A. JNDI (Java Naming and Directory Interface) is a J2EE lookup service that maps names to services and resources. JNDI provides a directory of advertised resources that exist on a particular stand-alone (unclustered) WebLogic server, or within a WebLogic server cluster. Examples of such resources include JMS connection factories, JMS destinations, JDBC (database) data sources, and application EJBs.

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

Q. What is a JMS connection factory?

A. A JMS connection factory is a named entity resource stored in JNDI. Applications, message driven beans (MDBs), and messaging bridges lookup a JMS connection factory in JNDI and use it to create JMS connections. JMS connections are used in turn to create JMS sessions, producers, and consumers that can send or receive messages.

Q. What is a JMS connection-id?

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

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

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

Q. What is a topic subscription?

A. A topic subscription can be thought of as an internal queue of messages waiting to be delivered to a particular subscriber. This internal queue accumulates copies of each message published to

the topic after the subscription was created. Conversely, it does not accumulate messages that were sent before the subscription was created. Subscriptions are not sharable, only one subscriber may subscribe to a particular subscription at a time.

Q. What is a non-durable topic subscriber?

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

Q. What is a durable subscriber?

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

Understanding Transactions

Q. What is a transaction?

A. A transaction is a set of distinct application operations that must be treated as an atomic unit. To maintain consistency, all operations in a transaction must either all succeed or all fail. See Introducing Transactions in *Programming WebLogic JTA*.

Q. Why are transactions important for integration?

A. Integration applications often use transactions to assure data consistency. For example, to assure that a message is forwarded exactly-once, a single transaction is often used to encompass the two operations of receiving the message from its source destination and sending to the target destination. Transactions are also often used to ensure atomicity of updating a database and performing a messaging operation.

Q. What is a JTA/XA/global transaction?

A. In J2EE, the terms JTA transaction, XA transaction, user transaction, and global transaction are often used interchangeably to refer to a single global transaction. Such a transaction may include operations on multiple different *XA capable* resources and even different resource types. A JTA transaction is always associated with the current thread, and may be passed from server to

server as one application calls another. A common example of an XA transaction is one that includes both a WebLogic JMS operation and a JDBC (database) operation.

Q. What is a local transaction?

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

Q. How does JMS provide local transactions?

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

Q. Are JMS local transactions useful for integration purposes?

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

Q. What is Automatic Transaction Enlistment?

A. Operations on resources such as database servers or messaging servers participate in a J2EE JTA transaction provided that:

- the resource is XA transaction capable

- the resource has been enlisted with the current transaction

- the client library used to access the resource is transaction aware (XA enabled).

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

- WebLogic clients using XA enabled WebLogic APIs automatically enlist operation in the current thread's JTA transaction. Examples of XA enabled WebLogic clients include WebLogic JMS XA enabled (or user transaction enabled) connection factories, and JDBC connection pool data sources that are global transaction enabled.

- Foreign (non-WebLogic) JMS clients do not automatically enlist in the current JTA transaction. Such clients must either go through an extra step of programmatically enlisting

in the current transaction, or use WebLogic provided features that wrap the foreign JMS client and automatically enlist when the foreign JMS client is accessed via wrapper APIs.

JMS features that provide automatic enlistment for foreign vendors are:

- Message-Driven EJBs

- JMS resource-reference pools

- Messaging Bridges

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

How to Integrate with a Remote Provider

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

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

1. Look up a JMS connection factory object and a JMS destination object using JNDI

2. Create a JMS connection using the connection factory object

3. Create message consumers or producers using the JMS connection and JMS destination objects.

Q. What information do I need to set up communications with a remote JMS provider?

A. You will need the following information to set up communications with a remote JMS provider:

- The destination type—whether the remote JMS destination is a queue or a topic.

- The JNDI name of the remote JMS destination.

- For durable topic subscribers—the connection-id and subscriber-id names that uniquely identify them. Message Driven EJBs provide default values for these values based on the EJB name.

- For non-WebLogic remote JMS providers

  – Initial Context Factory Class Name— the java class name of the remote JMS Provider's JNDI lookup service.

    – The file location of the java jars containing the remote JMS provider's JMS client and JNDI client libraries. Ensure that these jars are specified in the local JVM's classpath.

- The URL of the remote provider's JNDI service. For WebLogic servers, the URL is normally in the form `t3://hostaddress:port`. If you are tunneling over HTTP, begin the URL with `http` rather than `t3`. No URL is required for server application code that accesses a WebLogic JMS server that resides on the same WebLogic server or WebLogic cluster as the application.

- The JNDI name of the remote provider's JMS connection factory. This connection factory must exist on the remote provider, not the local provider.

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

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

  – `weblogic.jms.ConnectionFactory`—a non-XA capable factory.

  – `weblogic.jms.XAConnectionFactory`—an XA-capable factory

  – `weblogic.jms.MessageDrivenBeanConnectionFactory`—an XA-capable factory for message driven EJBs.

  Additional WebLogic JMS connection factories must be explicitly configured.

Q. What if a foreign JMS provider JNDI service has limited functionality?

A. The preferred method for locating JMS provider connection factories and destinations is to use a standard J2EE JNDI lookup. Occasionally a non-WebLogic JMS provider's JNDI service is hard to use or unreliable. The solution is to create a startup class or load-on-start servlet that runs on a WebLogic server that does the following:

    – Uses the foreign provider's proprietary (non-JNDI) APIs to locate connection factories and JMS destinations.

    – Registers the JMS destinations and JMS connection factories in WebLogic JNDI.

For sample code, see Creating Foreign JNDI Objects in a Startup Class in *Using Foreign JMS Providers with WebLogic Server* on Dev2Dev.

Q. How can I pool JMS resources?

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

pooled. For more information on resource pooling, including information on writing a custom pool, see the BEA *WebLogic JMS Performance Guide* on Dev2Dev..

Q. What version interoperability does WebLogic provide?

A. All WebLogic server releases 6.1 and higher interoperate freely between releases. For example, a WebLogic 8.1 JMS client can send messages directly to a 6.1 JMS server and vice versa. A Messaging Bridge can be used to forward WebLogic 5.1 JMS messages to and from WebLogic server releases 6.1 and higher.

Q. What tools are available for integrating with remote JMS providers?

A. The following table summarizes the tools available for integrating with remote JMS providers:

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

Best Practices when Integrating with Remote Providers

Q. How do I receive messages from a remote a JMS provider from within an EJB or Servlet?

A. Use a message driven EJB. Synchronous receives are not recommended because they idle a server side thread while the receiver blocks waiting for a message. See Q. What is a Message Driven EJB (MDB)? and Q. When should I use a MDB?

Q. How do I send messages to a remote JMS provider from within an EJB or Servlet?

A. Use a resource reference. It provides pooling and automatic enlistment. See Q. What are JMS resource references? and Q. What advantages do JMS resource references provide? In limited cases where wrappers are not sufficient, you can write your own pooling code. See the BEA WebLogic JMS Performance Guide on Dev2Dev.

If the target destination is remote, consider adding a local destination and messaging bridge to implement a store-and-forward high availability design. See Q. What is a messaging bridge? and Q. When should I use a messaging bridge?.

Another best practice is to use foreign JMS server definitions. Foreign JMS server definitions allow an application's JMS resources to be administratively changed and avoid the problem of hard-coding URLs into application code. In addition, foreign JMS server definitions are required to enable resource references to reference remote JMS providers. See Q. What are Foreign JMS Server Definitions? and Q. When is it best to use a Foreign JMS Server Definition?.

Q. How do I communicate with remote JMS providers from a client?

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

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

- Q. What is a messaging bridge?

- Q. When should I use a messaging bridge?

- Using WebLogic Store-and-Forward

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

Q. How can I tune WebLogic JMS interoperability features?

A. See Tuning WebLogic Server EJBs, Tuning WebLogic Message Bridge, and Tuning WebLogic JMS Store-and-Forward in *WebLogic Server Performance and Tuning*.

Using Foreign JMS Server Definitions

Q. What are Foreign JMS Server Definitions?

A. Foreign JMS server definitions are an administratively configured symbolic link between a JNDI object in a remote JNDI directory, such as a JMS connection factory or destination object, and a JNDI name in the JNDI name space for a stand-alone WebLogic Server or a WebLogic cluster. They can be configured using the Administration console, standard JMX MBean APIs, or programmatically using scripting. See Simple Access to Remote or Foreign JMS Providers in the Administration Console Online Help.

Q. When is it best to use a Foreign JMS Server Definition?

A. For this release, a Foreign JMS Server definition conveniently moves JMS JNDI parameters into one central place. You can share one definition between EJBs, servlets, and messaging bridges. You can change a definition without recompiling or changing deployment descriptors. They are especially useful for:

- Any message driven EJB (MDB) where it is desirable to administer standard JMS communication properties via configuration rather than hard code them into the application's EJB deployment descriptors. This applies even if the MDB's source destination isn't remote.

- Any MDB that has a destination remote to the cluster. This simplifies deployment descriptor configuration and enhances administrative control.

- Any EJB or servlet that sends or receives from a remote destination.

- Enabling resource references to refer to remote JMS providers. See Q. What are JMS resource references? and Q. How do I use resource references with non-transactional messaging?.

Using EJB/Servlet JMS Resource References

Q. What are JMS resource references?

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

JMS resource-references provide two additional features:

- Automatic pooling of JMS resources when those resources are closed by the application.

- Automatic enlistment of JMS resources with the current transaction, even for non-WebLogic JMS providers.

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

Resource references provide no functionality outside of application code, and therefore are not useful for configuring a message driven EJB's source destination or a messaging bridge's source or target destinations.

For WebLogic documentation on JMS resource-reference pooling, see Using JMS with EJBs and Servlets in *Programming WebLogic JMS*.

Q. What advantages do JMS resource references provide?

A. JMS resource references provide the following advantages:

- They ensure portability of servlet and EJB applications: they can be used to change an application's JMS resource without recompiling the application's source code.

- They provide automatic pooling of JMS Connection, Session, and MessageProducer objects.

- They provide automatic transaction enlistment for non-WebLogic JMS providers. This requires XA support in the JMS provider. If resource references aren't used, then enlisting a non-WebLogic JMS provider with the current transaction requires extra programmatic steps.

Q. How do I use resource references with foreign JMS providers?

A. To enable resource references to reference remote JMS providers, they must be used in conjunction with a foreign JMS definition. This is because resources references do not provide a place to specify a URL or initial context factory. See Q. What are Foreign JMS Server Definitions?.

Q. How do I use resource references with non-transactional messaging?

A. For non-transactional cases, do not use a global transaction (XA) capable connection factory. This will impact messaging performance. If you do, the resource reference will automatically begin and commit an internal transaction for each messaging operation. See Q. What is a transaction?.

Using WebLogic Store-and-Forward

Q. What is the WebLogic Store-and-Forward Service?

A. The WebLogic Store-and-Forward (SAF) Service enables WebLogic Server to deliver messages reliably between applications that are distributed across WebLogic Server instances. For example, with the SAF service, an application that runs on or connects to a local WebLogic Server instance can reliably send messages to a destination that resides on a remote server. If the destination is not available at the moment the messages are sent, either because of network problems or system failures, then the messages are saved on a local server instance, and are forwarded to the remote destination once it becomes available. See Understanding the Store-and-Forward Service in *Configuring and Managing WebLogic Store-and-Forward*.

Q. When should I use the WebLogic Store-and-Forward Service?

A. The WebLogic Store-and-Forward (SAF) Service should be used when forwarding JMS messages between WebLogic Server 9.0 or later domains. The SAF service can deliver messages:

- Between two stand-alone server instances.

- Between server instances in a cluster.

- Across two clusters in a domain.

- Across separate domains.

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

A. You can't use the WebLogic Store-and-Forward service in the following situations:

- Receiving from a remote destination—use a message driven EJB or implement a client consumer directly.

- Sending messages to a local destination—send directly to the local destination.

- Forwarding messages to prior releases of WebLogic Server. See Q. What is a messaging bridge?.

- Interoperating with third-party JMS products (for example, MQSeries). See Q. What is a messaging bridge?.

- When using temporary destinations with the `JMSReplyTo` field to return a response to a request.

- Environment with low tolerance for message latency. SAF increases latency and may lower throughput.

Using WebLogic JMS SAF CLient

Q. What is the WebLogic JMS SAF Client?

A. The JMS SAF Client feature extends the JMS store-and-forward service introduced in WebLogic Server 9.0 to standalone JMS clients. Now JMS clients can reliably send messages to server-side JMS destinations, even when the client cannot reach a destination (for example, due to a temporary network connection failure). While disconnected from the server, messages sent by a JMS SAF client are stored locally on the client file system and are forwarded to server-side JMS destinations when the client reconnects. See Reliably Sending Messages Using the JMS SAF Client.

Q. When should I use JMS SAF?

A. To provide store-and-forward high availability for JMS messages for WebLogic 9.2 and higher domains.

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

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

Q. What are the limitations of using the JMS SAF Client?

A. See Limitations of Using the JMS SAF Client.

Using a Messaging Bridge

Q. What is a messaging bridge?

A. Messaging bridges are administratively configured services that run on a WebLogic server. They automatically forward messages from a configured source JMS destination to a configured target JMS destination. These destinations can be on different servers than the bridge and can even be foreign (non-WebLogic) destinations. Each bridge destination is configured using the four common properties of a remote provider:

- The initial context factory.

- The connection URL.

- The connection factory JNDI name.

- The destination JNDI name.

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

Q. When should I use a messaging bridge?

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

Q. When should I avoid using a messaging bridge?

A. Other methods are preferred in the following situations:

- Receiving from a remote destination—use a message driven EJB or implement a client consumer directly.

- Sending messages to a local destination—send directly to the local destination.

- Environment with low tolerance for message latency. Messaging Bridges increase latency and may lower throughput. Messaging bridges increase latency for messages as they introduce an extra destination in the message path and may lower throughput because they forward messages using a single thread.

- Forward messages between WebLogic 9.x and higher domains—Use WebLogic Store-and-Forward. See Q. What is the WebLogic Store-and-Forward Service?.

Using Messaging Beans

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

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

- Automatically connects to a source destination and automatically retries connecting if the remote destination is inaccessible.

- Support automatic enlistment of the received messages in container managed transactions, even when the JMS provider is not WebLogic.

- Automatically pool their internal JMS connections, sessions, and consumers.

- A MDB's source destination, URL, and connection factory are configured in the EJB and WebLogic descriptors which are packaged as part of an application.

- The messaging processing application logic is contained in a single method callback `onMessage()`.

- AMDB is a full-fledged EJB that supports transactions, security, JDBC, and other typical EJB actions.

For more information, see Message-Driven EJBs in *Programming WebLogic Enterprise JavaBeans*.

Q. When should I use a MDB?

A. MDBs are the preferred mechanism for WebLogic server applications that receive and process JMS messages.

Q. Do I need to use a Messaging Bridge with a MDB?

A. Configure MDBs to directly consume from their source destination rather than insert a messaging bridge between them. MDBs automatically retry connecting to their source destination if the source destination is inaccessible, so there is no need to insert a messaging bridge in the message path to provide higher availability. Introducing a messaging bridge may have a performance impact. See Q. When should I avoid using a messaging bridge?.

Q. What is the best way to configure a MDB?

A. The following section provides tips for configuring a MDB:

- To configure MDB concurrency and thread pools, use the `max-beans-in-free-pool` and `dispatch-policy` descriptor fields. WebLogic may create fewer concurrent instances than `max-beans-in-free-pool` depending on the number of available server threads in the MDB's thread pool.

- Use foreign JMS server definitions when configuring a MDB to consume from a remote JMS provider. Although WebLogic MDB descriptors can be configured to directly refer to remote destinations, this information is packaged with the application and is not dynamically editable. You should configure a foreign JMS server definition and then configure the MDB to reference the foreign definition instead. Please note that some documentation refers to foreign JMS server definitions as wrappers. See Q. What are Foreign JMS Server Definitions?.

- Use care when configuring a MDB for container managed transactions. A MDB supports container managed XA transactions when a MDB's descriptor files have `transaction-type` of `Container` and a `trans-attribute` of `Required` and the JMS

connection factory is XA enabled. Failure to follow these steps will result in the MDB being non-transactional. The default WebLogic setting for a MDB connection factory is XA enabled. The MDB automatically begins a transaction and automatically enlists the received message in the transaction.

JMS Interoperability Resources

Q. What additional resources document JMS interoperability?

A. For general information on WebLogic JMS see the Messaging topic page and featured JMS articles at `http://dev2dev.bea.com/jms/`.

FAQs: Integrating Remote JMS Providers