BEA WebLogic Server

Programming WebLogic JMS

BEA WebLogic Server Version 6.0
Document Edition 1.0
December 2000

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic JMS

Document Edition	Date	Software Version
1.0	December 2000	6.0

Contents

About This Document

1. Introduction to WebLogic JMS

2. WebLogic JMS Fundamentals

3. Managing WebLogic JMS

4. Developing a WebLogic JMS Application

5. Using Transactions with WebLogic JMS

6. Migrating WebLogic JMS Applications

A. Configuration Checklists

B. JDBC Database Utility

Index

About This Document

This document explains how to use the BEA WebLogic Server™ platform to implement the Java™ Messaging Service (JMS) API for accessing enterprise messaging systems.

The document is organized as follows:

- Chapter 1, “Introduction to WebLogic JMS,”provides an overview of WebLogic Java Message Service (JMS).
- Chapter 2, “WebLogic JMS Fundamentals,” describes WebLogic JMS components and features.
- Chapter 3, “Managing WebLogic JMS,” provides an overview of configuring and monitoring WebLogic JMS.
- Chapter 4, “Developing a WebLogic JMS Application,” describes how to develop a WebLogic JMS application.
- Chapter 5, “Using Transactions with WebLogic JMS,” describes how to use transactions with WebLogic JMS.
- Chapter 6, “Migrating WebLogic JMS Applications,” describes how to migrate WebLogic JMS applications.
- Chapter A, “Configuration Checklists,” provides monitoring checklists for various WebLogic JMS features.
- Chapter B, “JDBC Database Utility,” describes how to use the the JDBC database utility to generate new JDBC stores and delete existing ones.

Audience

This document is written for application developers who want to design, develop, configure, and manage JMS applications using the Java 2 Platform, Enterprise Edition (J2EE) from Sun Microsystems. It is assumed that readers know JMS, JNDI (Java Naming and Directory Interface), the Java programming language, the Enterprise JavaBeans™ (EJB™), and Java Transaction API (JTA) of the J2EE specification.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation. Or you can go directly to the WebLogic Server Product Documentation page at http://e-docs.bea.com/wls/docs60.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

The BEA corporate Web site provides all documentation for WebLogic Server. For more information on JMS, access the JMS Javadoc and the JMS API – Errata, supplied on the Sun Microsystems Javasoft Web site at the follow locations:

http://www.java.sun.com/products/jms/javadoc-102a/index.html

http://www.java.sun.com/products/jms/errata_051801.html

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`

Convention	Usage

|	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

. . .	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

1 Introduction to WebLogic JMS

The following sections provide an overview of the WebLogic Java Messaging Service (JMS):

- What Is JMS?
- WebLogic JMS Features
- WebLogic JMS Architecture
- WebLogic JMS Extensions

What Is JMS?

An enterprise messaging system, also referred to as Message-Oriented Middleware (MOM), 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.

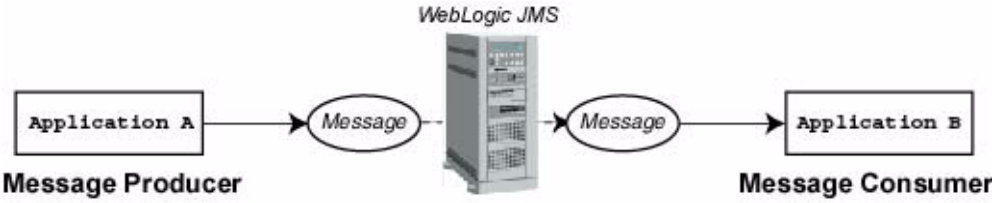
The Java Message Service (JMS) is a standard API for accessing enterprise messaging systems. Specifically, JMS:

- 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 1-1 WebLogic JMS Messaging



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

WebLogic JMS Features

WebLogic JMS provides a full implementation of the JMS API. Specifically, WebLogic JMS:

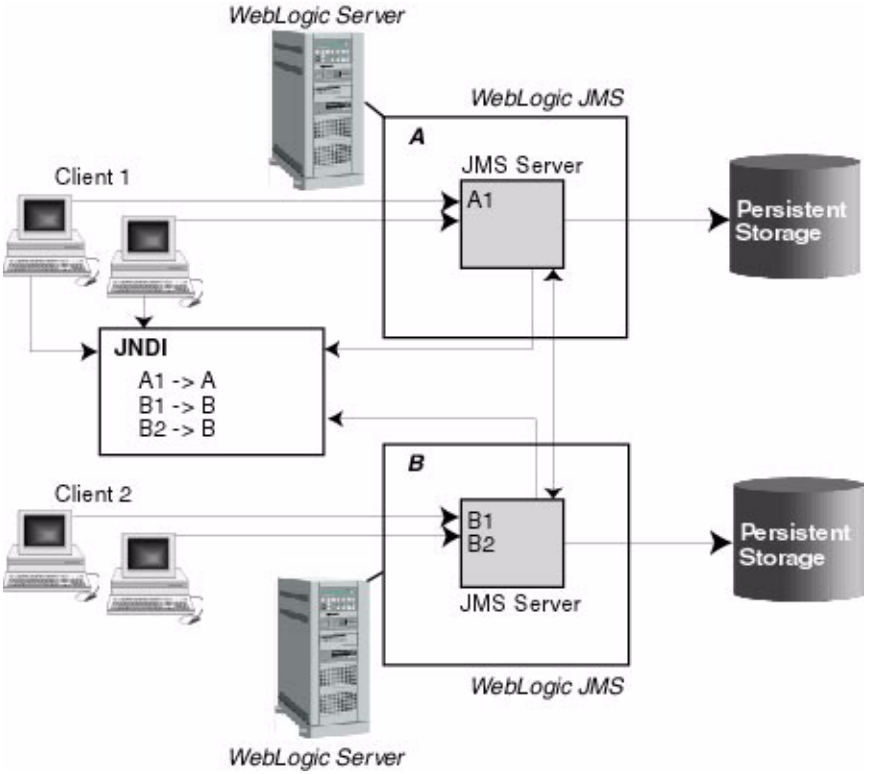
- Provides a single, unified messaging API.
- Implements the JavaSoft JMS specification version 1.0.2a, including the latest JMS API – Errata.
- Support clustering.
- Supports messaging for applications that span different operating systems and machine architectures.
- Can be configured by setting attributes from the WebLogic Administration Console and/or using the JMS API to override values.
- Allows interoperability between JMS applications and other resource managers (primarily databases) using the Java Transaction API (JTA) transactions. JMS applications can participate in transactions with other Java APIs that use JTA.

- Supports messages containing Extensible Markup Language (XML).
- Supports multicasting allowing the delivery of messages to a select group of hosts using an IP multicast address.
- May use either a database or file for persistent message storage.
- Can be used with other BEA WebLogic Server™ APIs and facilities, such as Enterprise Java Beans (EJB), JDBC connection pools, Servlets, and RMI.

WebLogic JMS Architecture

The following figure illustrates the WebLogic JMS architecture.

Figure 1-2 WebLogic JMS Architecture



Major Components

The major components of the WebLogic JMS Server architecture, as illustrated in the figure "WebLogic JMS Architecture" on page 1-4, include:

- WebLogic JMS servers implementing the messaging facility
- Client applications
- JNDI (Java Naming and Directory Interface), which provides a server *lookup* facility
- Backing stores (file or database) for storing persistent data

Clustering Features

The WebLogic JMS architecture implements *clustering* of multiple JMS servers. For detailed information about WebLogic clustering, see the *Using WebLogic Server Clusters*.

The advantages of clustering include:

- *Load balancing of destinations across multiple servers in the cluster*

  A system administrator can establish load balancing of destinations across multiple JMS servers in the cluster by configuring multiple JMS servers and using *targets* to assign them to the defined WebLogic Servers. Each JMS server is deployed on exactly one WebLogic Server and handles requests for a set of destinations.

  Note: Load balancing is not dynamic. During the configuration phase, the system administrator defines load balancing by specifying targets for JMS servers.

- *Cluster-wide, transparent access to destinations from any server in the cluster*

  A system administrator can establish cluster-wide, transparent access to destinations from any server in the cluster by configuring multiple connection factories and using *targets* to assign them to WebLogic Servers. Each connection factory can be deployed on multiple WebLogic Servers.

The application uses the Java Naming and Directory Interface (JNDI) to look up a connection factory and create a connection to establish communication with a JMS server. Each JMS server handles requests for a set of destinations. Requests for destinations not handled by a JMS server are forwarded to the appropriate server.

Connection factories are described in more detail in "WebLogic JMS Fundamentals" on page 2-1.

- *Scalability*

  Scalability is provided by:

  - Load balancing of destinations across multiple servers in the cluster, as described previously.
  - Distribution of application load across multiple JMS servers via connection factories, thus reducing the load on any single JMS server and enabling session concentration by routing connections to specific servers.
  - Optional multicast support, reducing the number of messages required to be delivered by a JMS server. The JMS server forwards only a single copy of a message to each host group associated with a multicast IP address, regardless of the number of applications that have subscribed.

Note: Automatic failover is not supported by JMS for this release.

WebLogic JMS Extensions

In addition to the API specified by the JavaSoft JMS specification version 1.0.2, WebLogic JMS provides a public API, `weblogic.jms.extensions`, that includes classes and methods for the extensions described in the following table.

Table 1-1 WebLogic JMS Extensions

Extension	For more information. . .
Create XML messages	Refer to "Step 6a: Create the Message Object (Message Producers)" on page 4-14
Define a session exception listener	Refer to "Defining a Session Exception Listener" on page 4-40
Set or display the maximum number of pre-fetched asynchronous messages allowed on the session	Refer to "Dynamically Configuring Multicasting Configuration Attributes" on page 4-77
Set or display the multicast session overrun policy that is applied when the message maximum is reached	Refer to "Dynamically Configuring Multicasting Configuration Attributes" on page 4-77
Dynamically create permanent queues or topics	Refer to "Using the JMSHelper Class Methods" on page 4-42
Convert between WebLogic JMS 6.0 and pre-6.0 `JMSMessageID` formats	Refer to "Setting Message Header Fields" on page 4-50

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 messages 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.

2 WebLogic JMS Fundamentals

The following sections describe WebLogic JMS components and features:

- Messaging Models
- WebLogic JMS Classes
- ConnectionFactory
- Connection
- Session
- Destination
- MessageProducer and MessageConsumer
- ServerSessionPoolFactory
- ServerSessionPool
- ServerSession
- ConnectionConsumer

Note: For more information on the JMS classes described in this section, access the JMS Javadoc, including the latest JMS API Errata, supplied on the Sun Microsystems Javasoft Web site at the following locations:
http://www.javasoft.com/products/jms/javadoc-102a/index.html
and
http://www.java.sun.com/products/jms/errata_051801.html

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.

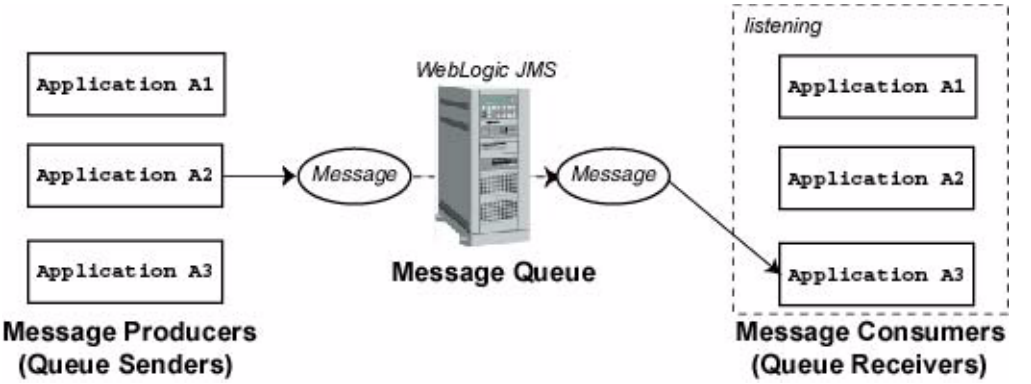
Note: The terms *producer* and *consumer* are used as generic descriptions of applications that send and receive messages, respectively, in either messaging model. For each specific messaging model, however, unique terms specific to that model are used when referring to producers and consumers.

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 2-1 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.

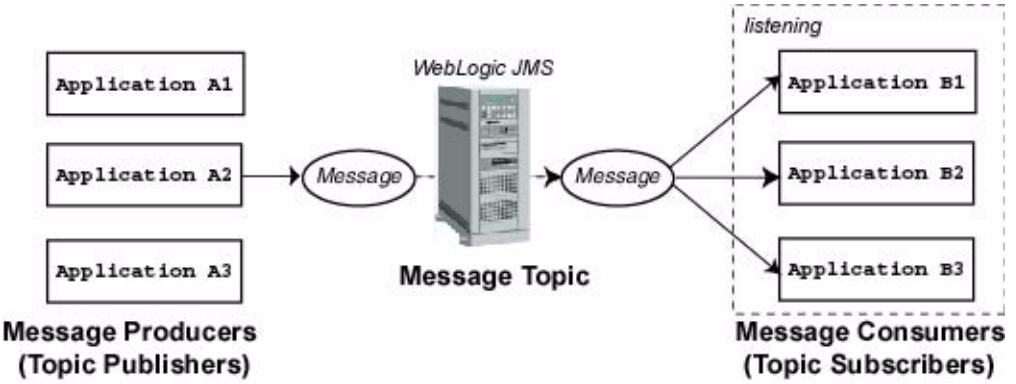
If multiple queue receivers are listening for messages on a queue, WebLogic JMS determines which one will receive the next message on a first come, first serve basis. If no queue receivers are listening on the queue, messages remain in the queue until a queue receiver attaches to the queue.

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



Unlike with the PTP messaging model, with the Pub/sub messaging model multiple topic subscribers can 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 4-46.

Message Persistence

Messages can be specified as persistent or non-persistent.

A persistent message is guaranteed to be delivered at least *once*—it is not considered sent until it has been safely written in the file or database. WebLogic JMS writes persistent messages to a persistent backing store (file or JDBC database) assigned to each JMS server during configuration.

Non-persistent messages are not stored. 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.

WebLogic JMS Classes

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`, `weblogic.jms.ServerSessionPoolFactory`, or `weblogic.jms.extensions` javadoc.

Table 2-1 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.
`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.

Table 2-1 JMS Classes (Continued)

JMS Class	Description
`ServerSessionPool`[1]	Provides a pool of server sessions that can be used to process messages concurrently for connection consumers.
`ServerSession`[1]	Associates a thread with a JMS session.
`ConnectionConsumer`[1]	Specifies a consumer that retrieves server sessions to process messages concurrently.

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

For information about configuring JMS objects, see "Managing WebLogic JMS" on page 3-1. The procedure for setting up a JMS application are presented in "Setting Up a JMS Application" on page 4-4.

ConnectionFactory

A `ConnectionFactory` object encapsulates connection configuration information, and enables JMS applications to create a `Connection`. A system administrator configures connection factories to create connections with predefined attributes.

A system administrator defines and configures one or more connection factories, and the WebLogic Server adds them to the JNDI space during startup. The application then retrieves a connection factory using WebLogic JNDI.

Notes: WebLogic JMS provides one connection factory by default. You only need to define a connection factory if the default provided by WebLogic JMS is not suitable for your application. For information on configuring connection factories, see "Managing JMS" in the *Administration Guide*.

Two deprecated connection factories are also supported, `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory` for backwards compatibility. For information on migrating to a new default or user-defined connection factory from a deprecated connection factory, refer to "Migrating WebLogic JMS Applications" on page 6-1.

The `ConnectionFactory` class does not define methods; however, its subclasses define methods for the respective messaging models.

A connection factory supports concurrent use, enabling multiple threads to access the object simultaneously.

The following table describes the `ConnectionFactory` subclasses.

Table 2-2 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 WebLogic JMS Application" on page 4-1, or the `javax.jms.ConnectionFactory` javadoc.

Connection

A `Connection` object 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 4-38.

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

The following table describes the `Connection` subclasses.

Table 2-3 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 WebLogic 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 the Connection.

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

The following table describes the Session subclasses.

Table 2-4 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 WebLogic 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 2-5 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.

Table 2-5 Acknowledge Modes Used for Non-Transacted Sessions (Continued)

Acknowledge Mode	Description
`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.
`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" on page 4-73. **Note:** You should avoid using this mode if your application cannot handle lost or duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.

Transacted Session

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

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

JMS can participate in distributed transactions with other Java services, such as EJB, that use the Java Transaction API (JTA). Transacted sessions do not support this capability as the transaction is restricted to accessing the messages associated with that session. For more information about using JMS with JTA, see "Using JTA User Transactions" on page 5-6.

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.

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

A destination supports concurrent use, enabling multiple threads to access the object simultaneously.

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

Table 2-6 Destination Subclasses

Subclass. . .	In Messaging Model. . .	Manages Messages For. . .
`Queue`	PTP	JMS PTP provider.
`TemporaryQueue`	PTP	JMS PTP provider, and exists for the duration of the JMS connection in which the messages are created. A temporary queue can be consumed only by the queue connection that created it.
`Topic`	Pub/sub	JMS Pub/sub provider.
`TemporaryTopic`	Pub/sub	JMS PTP 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 "Browsing Header and Property Fields" on page 4-57.

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

MessageProducer and MessageConsumer

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

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

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

Table 2-7 MessageProducer and MessageConsumer Subclasses

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

The PTP model, as shown in the figure "Point-to-Point (PTP) Messaging" on page 2-3, allows multiple sessions to receive messages from the same queue. However, a message can only be delivered to one queue receiver. When there are multiple queue receivers, WebLogic JMS defines the next queue receiver that will receive a message on a first-come, first-serve basis.

The Pub/sub model, as shown in the figure "Publish/Subscribe (Pub/Sub) Messaging" on page 2-4, allows messages to be delivered to multiple topic subscribers. Topic subscribers can be durable or non-durable, as described in "Setting Up Durable Subscriptions" on page 4-46.

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 4-11.

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

Message

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

Message Header Fields

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

For information about setting message header fields, see "Setting and Browsing Message Header and Property Fields" on page 4-50, 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 2-8 Message Header Fields

Field	Description	Defined By
`JMSCorrelationID`	Specifies one of the following: a WebLogic `JMSMessageID` (described later in this table), an application-specific string, or a `byte[]` array. The `JMSCorrelationID` is used to correlate messages. There are two common applications for this field. The first application is to link messages by setting up a request/response scheme, as follows: 1. When an application sends a message, it stores the `JMSMessageID` value assigned to it. 2. When an application receives the message, it copies the `JMSMessageID` into the `JMSCorrelationID` field of a response message that it sends back to the sending application. The second application is to use the `JMSCorrelationID` field to carry any String you choose, enabling a series of messages to be linked with some application-determined value. All `JMSMessageIDs` start with an `ID:` prefix. If you use the `JMSCorrelationID` for some other application-specific string, it *must not* begin with the `ID:` prefix. **Note:** The `byte[] JMSCorrelationID` is available for external JMS providers and is not supported by WebLogic JMS. Calling `setJMSCorrelationIDAsBytes()` throws a `java.lang.UnsupportedOperationException`.	Application

Table 2-8 Message Header Fields (Continued)

Field	Description	Defined By
`JMSDeliveryMode`	Specifies `PERSISTENT` or `NON_PERSISTENT` messaging. When a persistent message is sent, WebLogic JMS stores it in the JMS file or JDBC database. The `send()` operation is not considered successful until delivery of the message can be guaranteed. A persistent message is guaranteed to be delivered at least once. WebLogic JMS does not store non-persistent messages in the JMS database. This mode of operation provides the lowest overhead. They are guaranteed to be delivered at least once unless there is a system failure, in which case messages may be lost. If a connection is closed or recovered, all non-persistent messages that have not yet been acknowledged will be redelivered. Once a non-persistent message is acknowledged, it will not be redelivered. When a message is sent, this value is ignored. When the message is received, it contains the delivery mode specified by the sending method.	`send()` method
`JMSDestination`	Specifies the destination (queue or topic) to which the message is to be delivered. The application's message producer sets the value of this field when the message is sent. When a message is sent, this value is ignored. When a message is received, its destination value must be equivalent to the value assigned when it was sent.	`send()` method
`JMSExpiration`	Specifies the expiration, or time-to-live value, for a message. WebLogic JMS calculates the `JMSExpiration` value as the sum of the application's time-to-live and the current GMT. If the application specifies time-to-live as 0, `JMSExpiration` is set to 0, which means the message never expires. WebLogic JMS removes expired messages from the system to prevent their delivery.	`send()` method
`JMSMessageID`	Contains a string value that uniquely identifies each message sent by a JMS Provider. All `JMSMessageID`s start with an `ID:` prefix. When a message is sent, this value is ignored. When the message is received, it contains a provider-assigned value.	`send()` method

Table 2-8 Message Header Fields (Continued)

Field	Description	Defined By
`JMSPriority`	Specifies the priority level. This field is set before a message is sent. JMS defines ten priority levels, 0 to 9, 0 being the lowest priority. Levels 0-4 indicate gradations of *normal* priority, and level 5-9 indicate gradations of *expedited* priority. When the message is received, it contains the value specified by the method sending the message. You can sort destinations by priority by configuring a destination key, as described in Managing JMS in the *Administration Guide*.	Message Consumer
`JMSRedelivered`	Specifies a flag set when a message is redelivered because no acknowledge was received. This flag is of interest to a receiving application only. If set, the flag indicates that JMS may have delivered the message previously because one of the following is true: ■ The application has already received the message, but did not acknowledge it. ■ The session's `recover()` method was called to restart the session beginning after the last acknowledged message. For more information about the `recover()` method, see "Recovering Received Messages" on page 4-33.	WebLogic JMS
`JMSReplyTo`	Specifies a queue or topic to which reply messages should be sent. This field is set by the sending application before the message is sent. This feature can be used with the `JMSCorrelationID` header field to coordinate request/response messages. Simply setting the `JMSReplyTo` field does not guarantee a response; it *enables* the receiving application to respond, if it so chooses. You may set the `JMSReplyTo` to null, which may have a semantic meaning to the receiving application, such as a notification event.	Application
`JMSTimeStamp`	Contains the time at which the message was sent. WebLogic JMS writes the timestamp in the message when it accepts the message for delivery, *not* when the application sends the message. When the message is received, it contains the timestamp. The value stored in the field is a Java millis time value.	Message Consumer

Table 2-8 Message Header Fields (Continued)

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

Message Property Fields

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

Although message property fields may be used for application-specific purposes, JMS provides them primarily for use in message selectors. For more information about message selectors, see “Filtering Messages” on page 4-58.

For information about setting message property fields, see “Setting and Browsing Message Header and Property Fields” on page 4-50, or to the `javax.jms.Message` javadoc.

Message Body

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

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

Table 2-9 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

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.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 4-61, or the `weblogic.jms.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 4-61, 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 4-61, 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 4-61, or the `javax.jms.ConnectionConsumer` javadoc.

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

3 Managing WebLogic JMS

The Administration Console provides the interface that you can use to enable, configure, and monitor the features of the WebLogic Server, including JMS. To invoke the Administration Console, refer the procedures described in *Administration Guide*.

The following sections provide an overview of configuring and monitoring WebLogic JMS:

- Configuring WebLogic JMS
- Configuring WebLogic JMS Clustering
- Monitoring WebLogic JMS
- Recovering From a WebLogic Server Failure

Configuring WebLogic JMS

Using the Administration Console, you define configuration attributes to:

- Enable JMS.
- Create JMS servers.
- Create and/or customize values for JMS servers, connection factories, destinations (queues and topics), destination templates, destination keys, backing stores, session pools, and connection consumers.
- Set up custom JMS applications.
- Define thresholds and quotas.
- Enable any desired JMS features, such as server clustering (see the next section), concurrent message processing, destination sort ordering, and persistent messaging.

WebLogic JMS provides default values for some configuration attributes; you must provide values for all others. If you specify an invalid value for any configuration attribute, or if you fail to specify a value for an attribute for which a default does not exist, the WebLogic Server will not boot JMS when you restart it. A sample JMS configuration is provided with the product.

When migrating from a previous release, the configuration information will be converted automatically, as described in “Migrating Existing Applications” on page 6-7.

Note: Appendix A, “Configuration Checklists,” provides checklists that enable you to view the attribute requirements and/or options for supporting various JMS features.

Configuring WebLogic JMS Clustering

A WebLogic *cluster* is a group of servers that, from the application point-of-view, operate as a single server. A cluster provides:

- Scalability—servers can be added to the cluster dynamically to increase capacity
- High-availability—redundancy of multiple servers insulates applications from failures

A *clustered service* is one that is available on multiple servers in the cluster. WebLogic JMS can be clustered by deploying the service on multiple WebLogic Servers. Each clustered service is represented by a replica-aware stub, which appears to the application as a normal RMI stub representing a single object, which in actuality represents a collection of replicas. Each stub employs a load algorithm that chooses which replica to call and that is transparent to the caller.

In order to use JMS in a clustered environment, you must:

1. Administer WebLogic clusters as described in "Configuring WebLogic Servers and Clusters" in the *Administration Guide*.
2. Identify server *targets* for JMS servers and connection factories using the Administration Console. For more information about the configuration attributes, see "Managing JMS" in the *Administration Guide*.

Note: You cannot deploy the same destination on more than one JMS server. In addition, you cannot deploy a JMS server on more than one WebLogic Server.

For more information about starting WebLogic clusters and its features and benefits, see the *Using WebLogic Server Clusters*.

Note: Automatic failover is not supported for the clustered JMS service for this release. For information about performing a manual failover, refer to "Recovering From a WebLogic Server Failure" on page 3-4.

Monitoring WebLogic JMS

Statistics are provided for the following JMS objects: JMS servers, connections, sessions, destinations, message producers, message consumers, and server session pools. You can monitor JMS statistics using the Administration Console.

JMS statistics continue to increment as long as the server is running. Statistics can only be reset when the server is rebooted.

For more information on configuring and monitoring WebLogic JMS, see "Managing JMS" in the *Administration Guide*.

Once WebLogic JMS has been configured, applications can begin sending and receiving messages through the JMS API, as described in "Developing a WebLogic JMS Application" on page 4-1.

Recovering From a WebLogic Server Failure

The procedures for recovering from a WebLogic Server failure, and performing a manual failover, including programming considerations, are described in detail in "Managing JMS" in the *Administration Guide*.

CHAPTER

4 Developing a WebLogic JMS Application

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

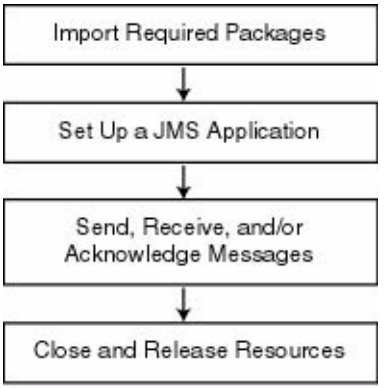
- Application Development Flow
- Importing Required Packages
- Setting Up a JMS Application
- Sending Messages
- Receiving Messages
- Acknowledging Received Messages
- Releasing Object Resources
- Managing Connections
- Managing Sessions
- Using Temporary Destinations
- Setting Up Durable Subscriptions
- Setting and Browsing Message Header and Property Fields
- Filtering Messages
- Defining Server Session Pools
- Using Multicasting

Note: For more information about the JMS classes described in this section, access the JMS Javadoc, including the latest Errata, supplied on the Sun Microsystems Javasoft Web site at the following locations:
http://www.javasoft.com/products/jms/Javadoc-102a/index.html
and
http://www.javasoft.com/products/jms/errata_051801.html

Application Development Flow

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

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



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

- Manage connection and session processing
- Create destinations dynamically
- Create durable subscriptions
- Manage message processing by setting and browsing message header and property fields, filtering messages, and/or processing messages concurrently

- Use multicasting
- Use JMS within transactions (described in "Using Transactions with WebLogic JMS" on page 5-1)

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

Importing Required Packages

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

Table 4-1 WebLogic JMS Packages

Package	Description
`javax.jms`	JavaSoft JMS API. This package is always used by WebLogic JMS applications.
`java.util`	Utility API, such as date and time facilities.
`java.io`	System input and output API.
`javax.naming` `weblogic.jndi`	JNDI packages required for server and destination lookups.
`javax.transaction.UserTransaction`	JTA API required for JTA user transaction support.
`weblogic.jms.ServerSessionPoolFactory`	WebLogic JMS public API for use with server session pools, an optional application server facility described in the JMS specification.
`weblogic.jms.extensions`	WebLogic-specific JMS public API that provides additional classes and methods, as described in "WebLogic JMS Extensions" on page 1-7.

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

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

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

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

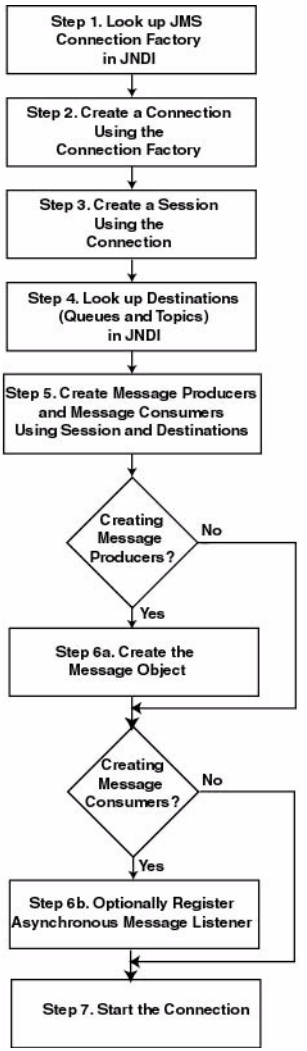
If you want to utilize any of the WebLogic JMS extension classes described in the previous table, also include the following statement on your import list:

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

Setting Up a JMS Application

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

Figure 4-2 Setting Up a JMS Application



The setup steps are described in the following sections. Detailed examples of setting up a Point-to-point (PTP) and Publish/subscribe (Pub/sub) application are also provided. The examples are excerpted from the `examples.jms` package provided with WebLogic Server in the `samples/examples` directory.

Before proceeding, ensure that the system administrator responsible for configuring WebLogic Server has configured the required JMS features, including the connection factories, JMS servers, and destinations. For more information, see "Managing JMS" in the *Administration Guide*.

For more information about the JMS classes and methods described in these sections, see "WebLogic JMS Classes" on page 2-5 or the `javax.jms` or `weblogic.jms.ServerSessionPoolFactory`, or `weblogic.jms.extensions` javadoc.

For information about setting up transacted applications and JTA user transactions, see "Using Transactions with WebLogic JMS" on page 5-1.

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 one default connection factory, that is included as part of the configuration by default. The WebLogic JMS system administrator may add or update connection factories during configuration. For information on configuring connection factories and the defaults that are available, see "Managing JMS" in the *Administration Guide*.

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

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

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

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

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

For more information about the `ConnectionFactory` class, see "ConnectionFactory" on page 2-6 or `javax.jms.ConnectionFactory` javadoc.

Step 2: Create a Connection Using the Connection Factory

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

For more information about the `Connection` class, see "Connection" on page 2-7 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 4-16.

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 4-16.

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

Step 3: Create a Session Using the Connection

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

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

For more information about the `Session` class, see "Session" on page 2-8 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 "Acknowledge Modes Used for Non-Transacted Sessions" on page 2-9. 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 2-9. 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 "Managing JMS" in the *Administration Guide*.

Once the destination has been configured, you can look up a destination by establishing a JNDI context (`context`), which has already been accomplished in "Step 1: Look Up a Connection Factory in JNDI" on page 4-6, 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 name defined during configuration.

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

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

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

The syntax for the `queueName` and/or `topicName` string is `JMS_Server_Name/Destination_Name` (for example, *`myjmsserver/mydestination`*). To view source code that uses this syntax, refer to the `findqueue()` example in "Creating Destinations Dynamically" on page 4-42.

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 "Creating Destinations Dynamically" on page 4-42.

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

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

```
public String getQueueName(
) throws JMSException

public String getTopicName(
) throws JMSException
```

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

For more information about the `Destination` class, see "Destination" on page 2-11 or `javax.jms.Destination` javadoc.

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. You can modify the message body by executing the corresponding message type's `reset()` 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 2-12, 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 4-58.

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 4-22.

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 "Setting Up Durable Subscriptions" on page 4-46. 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 `noLocal` flag (described later in this section). Message selectors are described in more detail in "Filtering Messages" on page 4-58.

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 4-22.

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 2-14, 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 4-16.

   If you wish to issue the `close()` method within an `onMessage()` method call, the system administrator must select the Allow Close In OnMessage checkbox when configuring the connection factory. For more information on configuring JMS, see "Managing JMS" in the *Administration Guide*.

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 Session Exception Listener" on page 4-40.

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 Session Exception Listener" on page 4-40.

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 2-12 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 4-38 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 `samples/examples/jms/queue` directory. 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 a servlet, use the following method:

```
InitialContext ic = newInitialContext();
```

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 2-8.

```
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 "WebLogic JMS Classes" on page 2-5 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 `samples/examples/jms/topic` directory. 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:

```
InitialContext ic = newInitialContext();
```

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 2-8.

```
    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 4-16.

For more information about the JMS classes used in this example, see “WebLogic JMS Classes” on page 2-5 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 4-4, you can send messages. To send a message, you must perform the following steps:

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

For more information about the JMS classes for sending messages and the message types, see the `javax.jms.Message` javadoc. For information about receiving messages, see "Receiving Messages" on page 4-29.

Step 1: Create a Message Object

This step has already been accomplished as part of the client setup procedure, as described in "Step 6a: Create the Message Object (Message Producers)" on page 4-14.

Step 2: Define a Message

This step *may* have been accomplished when setting up an application, as described in "Step 6a: Create the Message Object (Message Producers)" on page 4-14. Whether or not this step has already been accomplished depends on the method that was called to create the message object. For example, for TextMessage and ObjectMessage types, when you create a message object, you have the option of defining the message when you create the message object.

If a value has been specified and you do not wish to change it, you can proceed to step 3.

If a value has not been specified or if you wish to change an existing value, you can define a value using the appropriate `set` method. For example, the method for defining the text of a `TextMessage` is as follows:

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

Note: Messages can be defined as null.

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

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

For more information about the methods used to define messages, see the `javax.jms.Session` javadoc.

Step 3: Send the Message to a Destination

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

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 `reset()` method to clear the existing contents and enable write permission.

For more information about the `MessageProducer` class, see "MessageProducer and MessageConsumer" on page 2-12 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 "Managing JMS" in the *Administration Guide*.
- Values specified using the message producer's set methods, as described in "Dynamically Configuring Message Producer Configuration Attributes" on page 4-27.

If you define the delivery mode as `PERSISTENT`, you should configure a backing store for the destination, as described in "Managing JMS" in the *Administration Guide*.

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 4-11.

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 "Managing JMS" in the *Administration Guide*.
- Values specified using the message producer's set methods, as described in "Dynamically Configuring Message Producer Configuration Attributes" on page 4-27.

If you define the delivery mode as `PERSISTENT`, you should configure a backing store, as described in "Managing JMS" in the *Administration Guide*.

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 4-12.

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.

Dynamically Configuring Message Producer Configuration Attributes

As described in the previous section, when sending a message, you can optionally specify the delivery mode, timeout, and time-to-live values. If not specified, the delivery mode, priority, and time-to-live attributes are set to the connection factory or destination override configuration attributes defined for the producer, as described "Managing JMS" in the *Administration Guide*.

Alternatively, you can set the delivery mode, timeout, and time-to-live values dynamically using the message producers set methods to override the configured values.

The following table lists the message producer set and get methods for each dynamically configurable attribute.

Note: The delivery mode, timeout, and time-to-live attribute settings can be overridden by the destination using the Delivery Mode Override, Priority Override, and Time To Live Override destination configuration attributes, as described in 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`

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

For more information about the `MessageProducer` class methods, see the `javax.jms.MessageProducer` javadoc.

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 `samples/examples/jms/queue` directory. The example shows the code required to create a `TextMessage`, set the text of the message, and send the message to a queue.

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

For more information about the `QueueSender` 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 `samples/examples/jms/topic` directory. The following example 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 4-4, 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. Guidelines for programming client servlet applications to receive messages are provided in "Receiving Messages with Client Servlets" on page 4-31.

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

- Message delivery attributes (delivery mode and sorting criteria) defined during configuration, as described in "Managing JMS" in the *Administration Guide*, or as part of the `send()` method, as described in "Sending Messages" on page 4-22.
- Destination sort order set using destination keys, as described in "Managing JMS" in the *Administration Guide*.

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 `reset()` 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 4-22.

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 4-15.

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

Receiving Messages Synchronously

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

```
public Message receive(
) throws JMSException

public Message receive(
  long timeout
) throws JMSException

public Message receiveNoWait(
) throws JMSException
```

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

The `MessageConsumer` class methods are inherited by the `QueueReceiver` and `TopicSubscriber` classes. For additional information about the `MessageConsumer` class methods, see the `javax.jms.MessageConsumer` javadoc.

Example: Receiving Messages Synchronously Within a PTP Application

The following example is excerpted from the `examples.jms.queue.QueueReceive` example, provided with WebLogic Server in the `samples/examples/jms/queue` directory. Rather than set a message listener, you would call `qreceiver.receive()` for each message. For example:

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

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

Example: Receiving Messages Synchronously Within a Pub/Sub Application

The following example is excerpted from the `examples.jms.topic.TopicReceive` example, provided with WebLogic Server in the `samples/examples/jms/topic` directory. Rather than set a message listener, you would call `tsubscriber.receive()` for each message.

For example:

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

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

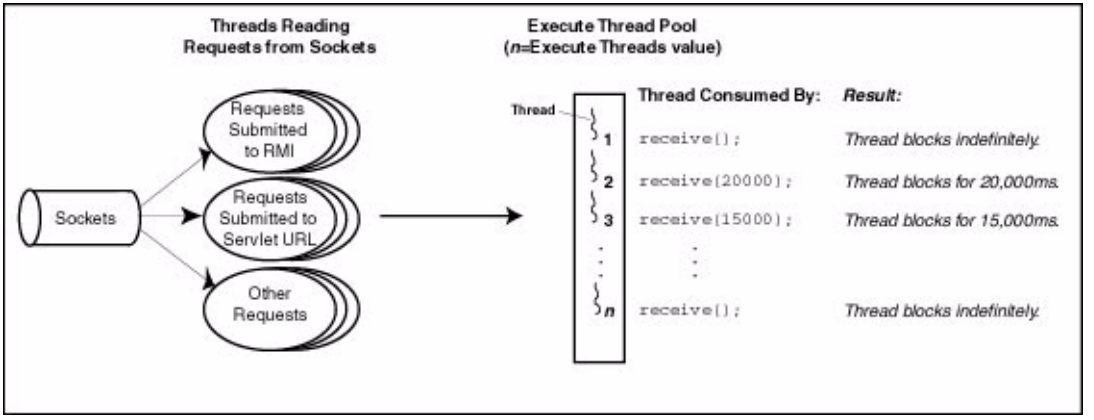
Receiving Messages with Client Servlets

The WebLogic Server execute thread pool has a finite number of threads available for processing requests. The number of threads available is defined by the system administrator when configuring the Server using the Execute Threads attribute. By default, 15 threads are available.

It is possible for all threads in the execute thread pool to be blocked causing a server deadlock, as illustrated in the following figure.

Note: Only blocking servlet JMS requests running in the servers JVM may potentially deadlock the server. RMI requests cannot cause a server to deadlock.

Figure 4-3 Example of a Deadlock in the Execute Thread Pool



As shown in the previous figure, once all configured threads from the execute thread pool are blocked, no threads are available for new requests. If the RMI requests and other pending servlet URLs are waiting for threads to become available to complete the blocking calls, the blocked calls never resume (because all available threads are blocked) and the server becomes deadlocked.

For example, if a `receive()` method is blocking a thread, and its corresponding `send()` method has been submitted to the RMI or servlet, the `send()` may not be able to access a thread in which to run. The `receive()` method blocks the thread until it receives its data from the `send()` method; the `send()` method cannot access a thread and execute because all of the threads are blocked, resulting in a deadlock.

To prevent a server deadlock, you should avoid the following:

- Coding servlets to receive messages synchronously and wait for a response.
- Specifying a long or infinite timeout value on the JMS `receive()` method.

Message polling is an inefficient use of resources. You should use a different, more efficient method for activating a thread when needed, and exit, rather than wait for future requirements, when the thread is no longer needed.

You should design the servlet applications to receive messages asynchronously using the `onMessage()` method, as described in "Receiving Messages Asynchronously" on page 4-30. In this way, no threads are wasted waiting for messages to be received.

Note: WebLogic JMS also implements an optional JMS facility for defining a server-managed pool of server sessions. This facility enables an application to process messages concurrently. For more information, see "Defining Server Session Pools" on page 4-61.

Server session pools can also be implemented using Message Driven Beans. For information on using message driven beans to implement server session pools, see to *Using WebLogic EJB*.

If your servlet program design requires messages to be received synchronously, it is recommended that you use one of the following methods, listed in order of preference:

- Use the `receiveNoWait()` method, which returns the next message or a null value if no message is currently available. In this case, the call does not block. The servlet should provide a way to return to or reschedule the request, without calling `wait()`.
- Pass a `timeout` value as an argument to the `receive()` method and set it to the minimum value (greater than 0) that is allowed by the application to avoid consuming threads that are waiting for a response from the server.

  Note: Use of this option should be minimized, as it may deadlock a busy server.

For more information about the methods described in this section, see "Receiving Messages" on page 4-29 or the `javax.jms` javadoc.

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 "Acknowledge Modes Used for Non-Transacted Sessions" on page 2-9. Synchronously received AUTO_ACKNOWLEDGE messages may not be received; 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

Messages in queues are not necessarily redelivered in the same order that they were originally delivered, nor to the same queue consumers.

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 "Acknowledge Modes Used for Non-Transacted Sessions" on page 2-9.

To acknowledge a received message, use the following `Message` method:

```
public void acknowledge(
) throws JMSException
```

The `acknowledge()` method acknowledges the current message and all previous messages received since the last client acknowledge. 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 and any outstanding synchronous applications are cancelled.
- 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 4-38 or "Closing a Session" on page 4-41, respectively.

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server in the `samples/examples/jms/queue` directory. This example shows the code required to close the message consumer, session, and connection objects.

```
public void close(
) throws JMSException
{
  qreceiver.close();
  qsession.close();
  qcon.close();
}
```

In the `QueueSend` example, the `close()` method is called at the end of `main()` to close objects and free resources.

Managing Connections

The following sections describe how to manage connections:

- Defining a Connection Exception Listener
- Accessing Connection Meta Data
- 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 Meta Data

You can access the meta data 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 meta data. The following table lists the various type of JMS meta data and the get methods that you can use to access them.

Table 4-3 Connection Meta Data Get Methods

JMS Meta Data	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 4-4. 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 5-6.
- 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 on Exception(
  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 4-30.

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 5-6.
- 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 wish to issue the `close()` method within an `onMessage()` method call, the system administrator must select the Allow Close In OnMessage checkbox when configuring the connection factory. For more information, see " JMS Connection Factories" in the Administration Console Online Help.

Creating Destinations Dynamically

You can create destinations dynamically using:

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

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

Using the JMSHelper Class Methods

You can dynamically submit an asynchronous request to create a queue or topic, respectively, using the following `JMSHelper` methods:

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

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

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

Each method updates the following:

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

Note: Either method call can fail without throwing an exception. In addition, a thrown exception does not necessarily indicate that the method call failed.

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

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

```
private static Queue findQueue (
  QueueSession queueSession,
  String jmsServerName,
  String queueName,
  int retryCount,
  long retryInterval
) throws JMSException
{
   String wlsQueueName = jmsServerName + "/" + queueName;
   String command = "QueueSession.createQueue(" +
       wlsQueueName + ")";
   long startTimeMillis = System.currentTimeMillis();
   for (int i=retryCount; i>=0; i--) {
       try {
         System.out.println("Trying " + command);
         Queue queue = queueSession.createQueue(wlsQueueName);
         System.out.println(command + "succeeded after " +
              (retryCount - i + 1) + " tries in " +
              (System.currentTimeMillis() - startTimeMillis) +
              " millis.");
         return queue;
       } catch (JMSException je) {
         if (retryCount == 0) throw je;
       }
       try {
         System.out.println(command + "> failed, pausing " +
              retryInterval + " millis.");
         Thread.sleep(retryInterval);
       } catch (InterruptedException ignore) {}

   }
   throw new JMSException("out of retries");
}
```

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

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

For more information on the `JMSHelper` class, refer to the `weblogic.jms.extensions.JMSHelper` javadoc.

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.

The WebLogic JMS server can use the `JMSReplyTo` header field to return a response to the application. The 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 4-45.

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

Note: Before creating a temporary destination (queue or topic), you must use the Adminstration Console to configure the JMS server to use temporary destinations. This is done by using the JMS Server’s `Temporary Template` attribute to select a JMS template that is configured in the same domain. For more information about configuring a JMS server, see “JMS Server” 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 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.

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 Client ID
- Creating Subscribers for a Durable Subscription
- Deleting Durable Subscriptions
- Modifying Durable Subscriptions

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 preferred method, according to the JMS specification, 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, see "JMS Connection Factories" in the Administration Console Online Help.

- Alternatively, an application 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. For information on the default connection factory, see "Managing JMS" in the *Administration Guide*.

  If a client ID is already defined for the connection, an `IllegalStateException` is thrown. If the specified client ID is already defined for another connection, an `InvalidClientIDException` is thrown.

  Note: When specifying the client ID using the `setClientID()` method, there is a risk that a duplicate client ID may be specified without throwing an exception. For example, if the client IDs for two separate connections are set simultaneously to the same value, a race condition may occur and the same value may be assigned to both connections. You can avoid this risk of duplication by specifying the client ID during configuration.

  To display a client ID and test whether or not a client ID has already been defined, use the following Connection method:

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

Note: Support for durable subscriptions is a feature unique to the Pub/sub messaging model, so client IDs are used only with topic connections; queue connections also contain client IDs, but JMS does not use them.

Durable subscriptions should not be created for a temporary topic, because a temporary topic is designed to exist only for the duration of the current connection.

Creating Subscribers for a Durable Subscription

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

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

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

You must specify the name of the topic for which you are creating a subscriber, and the name of the durable subscription. You may also specify a message selector for filtering messages and a `noLocal` flag (described later in this section). Message selectors are described in more detail in "Filtering Messages" on page 4-58. 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 4-46.

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

Deleting Durable Subscriptions

To delete a durable subscription, you use the following `TopicSession` method:

```
public void unsubscribe(
  String name
) throws JMSException
```

You must specify the name of the durable subscription to be deleted.

You cannot delete a durable subscription if any of the following are true:

- A `TopicSubscriber` is still active on the session.
- A message received by the durable subscription is part of a transaction or has not yet been acknowledged in the session.

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 4-49.

   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 4-48 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.

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. 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 4-58.

Setting Message Header Fields

JMS messages contain a standard set of header fields that are always transmitted with the message. They are available to message consumers that receive messages, and some fields can be set by the message producers that send messages. Once a message is received, its header field values can be modified.

For a description of the standard messages header fields, see "Message Header Fields" on page 2-14.

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 4-4 Message Header Set and Get Methods

Header Field	Set Method	Get Method

`JMSCorrelationID`	`public void setJMSCorrelationID( String correlationID ) throws JMSException`
	
	Note: The `byte[] JMSCorrelationID` is available for external JMS providers and is not supported by WebLogic JMS. Calling `setJMSCorrelationIDAsBytes()` throws a `java.lang.UnsupportedOperationException`.
		`public String getJMSCorrelationID( ) throws JMSException`
		
		`public byte[] getJMSCorrelationIDAsBytes( ) throws JMSException`

`JMSDestination`[1]	`public void setJMSDestination( Destination destination ) throws JMSException`	`public Destination getJMSDestination( ) throws JMSException`
`JMSDeliveryMode`[1]	`public void setJMSDeliveryMode( int deliveryMode ) throws JMSException`	`public int getJMSDeliveryMode( ) throws JMSException`
`JMSExpiration`[1]	`public void setJMSExpiration( long expiration ) throws JMSException`	`public long getJMSExpiration( ) throws JMSException`

Table 4-4 Message Header Set and Get Methods (Continued)

Header Field	Set Method	Get Method

JMSMessageID[1]	`public void setJMSMessageID( String id ) throws JMSException`
	In addition to the set method, the `weblogic.jms.extensions.JMSHelper` class provides the following methods to convert between WebLogic JMS 6.0 and pre-6.0 `JMSMessageID` formats:
	`public void oldJMSMessageIDToNew( String id, long timeStamp ) throws JMSException`
	`public void newJMSMessageIDToOld( String id, long timeStamp ) throws JMSException`
		`public String getJMSMessageID( ) throws JMSException`

JMSPriority[1]	`public void setJMSPriority( int priority ) throws JMSException`	`public int getJMSPriority( ) throws JMSException`
JMSRedelivered[1]	`public void setJMSRedelivered( boolean redelivered ) throws JMSException`	`public boolean getJMSRedelivered( ) throws JMSException`
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 `samples/examples/jms/sender` directory, 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 meta data contains a list of JMSX properties, which can be accessed as an enumerated list using the `getJMSXPropertyNames()` method. For more information, see "Accessing Connection Meta Data" on page 4-37.

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 4-5 Message Property Set and Get Methods for Data Types

Data Type	Set Method	Get Method
boolean	`public void setBooleanProperty( String name, boolean value ) throws JMSException`	`public boolean getBooleanProperty( String name ) throws JMSException`
byte	`public void setByteProperty( String name, byte value ) throws JMSException`	`public byte getByteProperty( String name ) throws JMSException`
double	`public void setDoubleProperty( String name, double value ) throws JMSException`	`public double getDoubleProperty( String name ) throws JMSException`
float	`public void setFloatProperty( String name, float value ) throws JMSException`	`public float getFloatProperty( String name ) throws JMSException`
int	`public void setIntProperty( String name, int value ) throws JMSException`	`public int getIntProperty( String name ) throws JMSException`

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

Data Type	Set Method	Get Method
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 4-6 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 2-18 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 4-58.

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 `samples/examples/jms/queue` directory, 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 4-35.

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 4-11.

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 4-50 and "Setting Message Property Fields" on page 4-53, 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)
```

You specify the syntax type, which for this release 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`

The methods return 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">
              <name>Hand-held Power Drill</name>
              <description>Compact, assorted colors.</description>
              <price>$34.99</price>
       <item id="123">
              <name>Mitre Saw</name>
              <description>Three blades sizes.</description>
              <price>$69.99</price>
       <item id="66">
              <name>Socket Wrench Set</name>
              <description>Set of 10.</description>
              <price>$19.99</price>
</order>
```

The following example shows how to retrieve the text node name of the second item in the previous example. The method call returns the following string: `Mitre Saw`.

```
JMS_BEA_SELECT('xpath', '/order/item[2]/name/text()');
```

The following example shows how to retrieve the ID attribute of the third item. This method call returns the following string: `66`.

```
JMS_BEA_SELECT('xpath', '/order/item[3]/attribute::id');
```

The following example shows how to retrieve all elements of item.

```
JMS_BEA_SELECT('xpath', '/order/item');
```

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.

Defining Server Session Pools

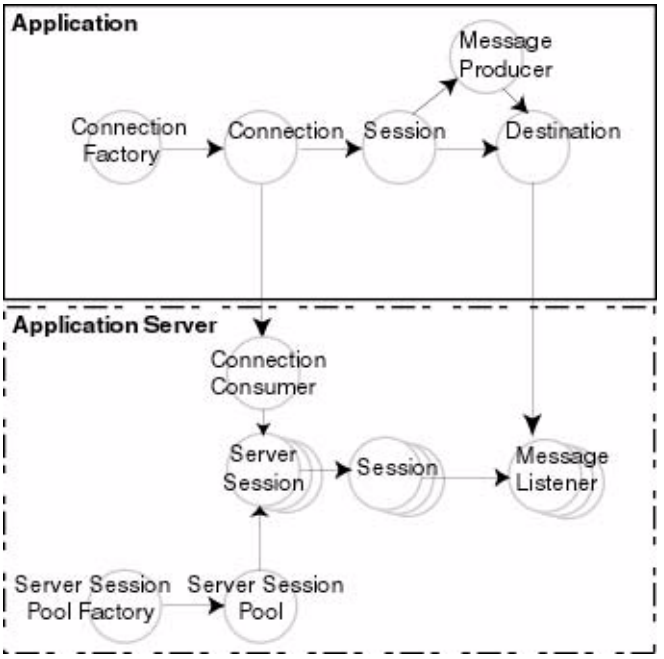
WebLogic JMS implements an optional JMS facility for defining a server-managed pool of server sessions. This facility enables an application to process messages concurrently.

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 4-4 Server Session Pool Facility

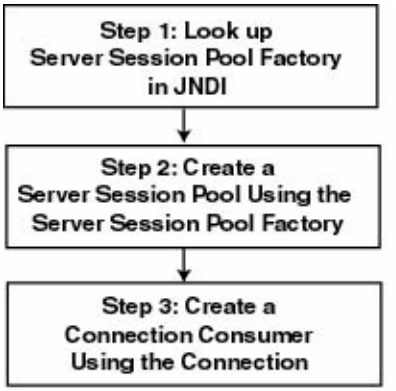


As illustrated in the figure, the application provides a single-threaded message listener. The connection consumer, implemented by JMS on the application server, performs the following tasks to process one or more messages:

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 4-5 Preparing for Concurrent Message Processing



Notes: When you create a server session pool (step 2), WebLogic Server first tests the `weblogic.allow.create.jms.ServerSessionPool` ACL to ensure the user has *create* permission. This permission is granted to *everyone*, by default. You can update this property to restrict the permission to a set of users and groups or you can delete the property to disable the server session pool feature. For more information about configuring ACLs, see "Managing Security" in the *Administration Guide*.

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 *Using WebLogic EJB*.

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 Managing JMS in the *Administration Guide*.

WebLogic JMS does *not* support the optional `TopicConnection.createDurableConnectionConsumer()` operation. For more information on this advanced JMS operation, refer to the JavaSoft JMS Specification version 1.0.2.

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.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 2-19 or `weblogic.jms.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 2-20 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.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.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 the "Managing JMS" in the *Administration Guide*
- Including in your application the Connection methods described in the following sections

For more information about the `ConnectionConsumer` class, see "ConnectionConsumer" on page 2-21 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 4-58.

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 4-58.

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 4-46.

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 4-16. 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.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.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 “WebLogic JMS Classes” on page 2-5 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 4-19. 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.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.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 “WebLogic JMS Classes” on page 2-5 or the `javax.jms` javadoc.

Using Multicasting

Multicasting enables the delivery of messages to a select group of hosts that subsequently forward the messages to subscribers.

The benefits of multicasting include:

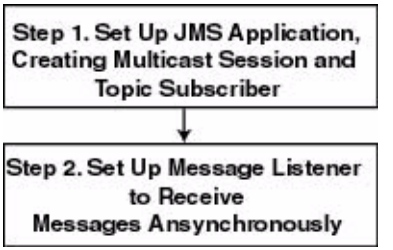
- Near real-time delivery of messages to host group
- High scalability due to the reduction in the amount of resources required by the JMS server to deliver messages to subscribers

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 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.

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

Figure 4-6 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.

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 4-77.
- For each destination, the multicast IP address, port, and time-to-live attributes are specified. To better understand the time-to-live attribute setting, see "Example: Multicast TTL" on page 4-78.

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 on the multicasting configuration attributes, see the Administration Console Online Help. The multicast configuration attributes are also summarized in Appendix A, "Configuration Checklists."

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

Set up the JMS application as described in "Setting Up a JMS Application" on page 4-4, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-8, 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 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
);
```

In addition, create a topic subscriber, as described in "Create TopicPublishers and TopicSubscribers" on page 4-12.

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.

Multicasting is only supported for non-durable subscribers. An attempt to create a durable subscriber on a multicast session will cause a `JMSException` to be thrown.

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.

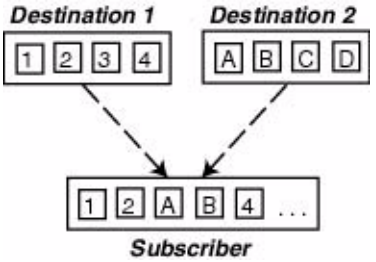
Set up the message listener for the topic subscriber, as described in "Receiving Messages Asynchronously" on page 4-30.

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 4-7 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 4-7 Message Producer Set and Get Methods

Attribute	Set Method	Get Method
Messages Maximum	`public void setMessagesMaximum( int messagesMaximum ) throws JMSException`	`public int getMessagesMaximum( ) throws JMSException`
Overrun Policy	`public void setOverrunPolicy ( int overrunPolicy ) throws JMSException`	`public int getOverrunPolicy( ) throws JMSException`

Note: The values set using the set methods take precedence over the configured values.

For more information about these `Session` class methods, see the `weblogic.jms.extensions.WLSession` javadoc. For more information on these multicast configuration attributes, see "JMS Destinations" in the Administration Console Online Help.

Example: Multicast TTL

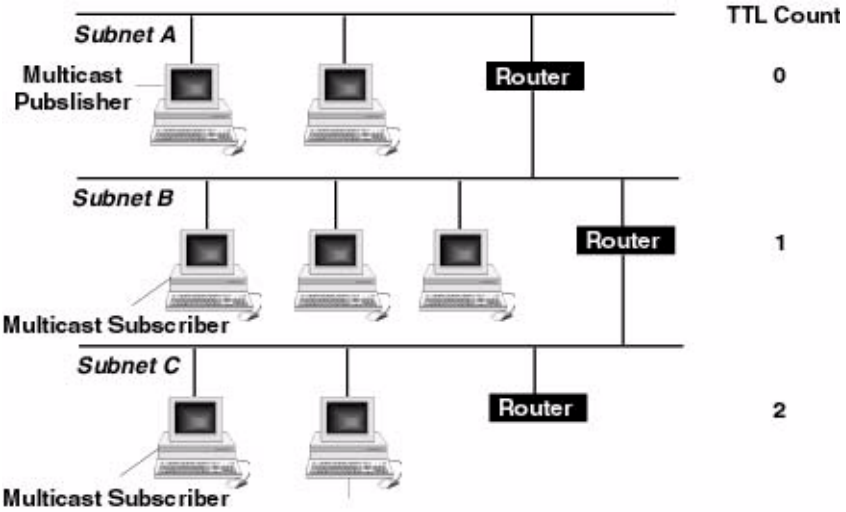
Note: The following example is a very simplified illustration of how the Multicast TTL (time-to-live) destination configuration attribute impacts the delivery of messages across routers. It is strongly advised that you seek the assistance of your network administrator when configuring the multicast TTL 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 on the multicast configuration attributes, see "JMS Destinations" in the Administration Console Online Help.

Consider the following network diagram.

Figure 4-8 Multicast TTL Example



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

If the Multicast TTL attribute is set to 0 (indicating that the messages cannot traverse any routers and are delivered on the current subnet only), when the multicast publisher on Subnet A publishes a message, the message will not be delivered to any of the multicast subscribers.

If the Multicast TTL attribute is set to 1 (indicating that messages can traverse one router), when the multicast publisher on Subnet A publishes a message, the multicast subscriber on Subnet B will receive the message.

Similarly, if the Multicast TTL attribute is set to 2 (indicating that messages can traverse two routers), when the multicast publisher on Subnet A publishes a message, the multicast subscribers on Subnets B and C will receive the message.

5 Using Transactions with WebLogic JMS

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

- Overview of Transactions
- Using JMS Transacted Sessions
- Using JTA User Transactions
- Asynchronous Messaging Within JTA User Transactions Using Message Driven Beans
- Example: JMS and EJB in a JTA User Transaction

Note: For more information about the JMS classes described in this section, access the JMS Javadoc, including the latest JMS API Errata, supplied on the Sun Microsystems Javasoft Web site at the following locations:
http://www.javasoft.com/products/jms/Javadoc-102a/index.html
and
http://www.javasoft.com/products/jms/errata_051801.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 5-6.

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 4-40.

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

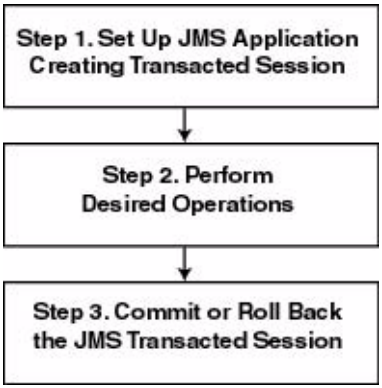
A JMS transacted session supports transactions that are located within the session. A JMS transacted session's transaction will not have any effects outside of the session. For example, rolling back a session will roll back all sends and receives on that session, but will not roll back any database updates.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

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

Figure 5-1 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 4-4, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-8, 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 rollback 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 rollback 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 *Programming WebLogic JTA*.

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

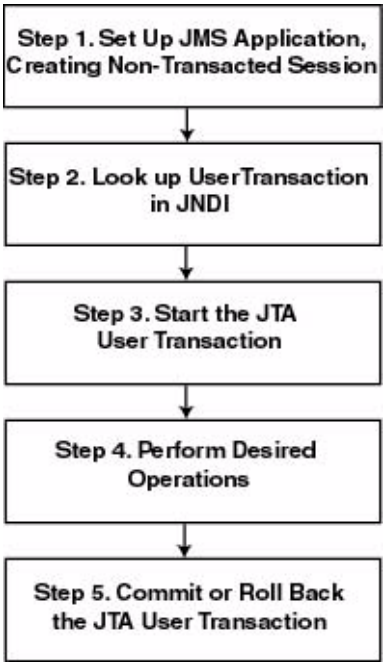
WebLogic Server supports the two-phase commit protocol (2PC), enabling an application to coordinate a single JTA transaction across two or more resource managers. It guarantees data integrity by ensuring that transactional updates are committed in all of the participating resource managers, or are fully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

Note: A separate 2PC transaction license is required to support this protocol. For transaction migration considerations related to 2PC, see "Migrating WebLogic JMS Applications" on page 6-1.

Before using a JTA transacted session, the system administrator must configure the connection factories to support JTA user transactions by selecting the User Transactions Enabled checkbox, as described in Managing JMS in the *Administration Guide*.

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

Figure 5-2 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 4-4, however, when creating sessions, as described in "Step 3: Create a Session Using the Connection" on page 4-8, 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 lookup 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 methods to commit or rollback the JTA user transaction.

To commit the transaction, execute the following 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 rollback the transaction, execute the following 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 to *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 User Transactions Enabled checkbox 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 `samples/examples/jta/jmsjdcb` directory

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 4-4.

```
//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()
```

6 Migrating WebLogic JMS Applications

The following sections describe how to migrate WebLogic JMS applications:

- Existing Feature Functionality Changes
- Migrating Existing Applications
- Deleting JDBC Database Stores

Existing Feature Functionality Changes

Changes in existing feature functionality have been made in order to comply with the JavaSoft JMS Specification version 1.0.2. and the latest JMS API – Errata.

The following table lists the changes in existing feature functionality from WebLogic Server Version 5.1, and also indicate any code changes that might be required as a result. For additional information pertaining to the Version 1.0.2 change history, see Chapter 11, “Change History,” of the JavaSoft JMS Specification Version 1.0.2.

Table 6-1 Existing Feature Functionality Changes

Category	Description	Code Modification

Connection Factories	The following two default connection factories have been deprecated: `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory.`
	These connection factories are still defined and usable in this release for backwards compatibility.
	WebLogic JMS 6.0 defines one connection factory, by default: `weblogic.jms.ConnectionFactory`
	You can also specify user-defined connection factories using the Administration Console.
	Note: Using the default connection factory, you have no control over the WebLogic server on which the connection factory may be deployed. If you would like to target a particular WebLogic Server, create a new connection factory and specify the appropriate WebLogic Server target(s).
		It is recommended that existing code that use the deprecated classes be modified to use a new default or user-defined connection factory class.
		For example, if your code specified the following constant using the default queue connection factory:
		`public final static String JMS_FACTORY="javax.jms.QueueConnectionFactory"`
		You should modify the constant to use a new user-defined connection factory, for example:
		`public final static String JMS_FACTORY="weblogic.jms.QueueConnectionFactory"`
		For true backwards compatibility with previous releases, you should ensure that you select the Allow Close In onMessage and User Transactions Enabled checkboxes when configuring the connection factory.
		For more information about defining connection factories, see "JMS Connection Factories" in the Administration Console Online Help.

	In order to instantiate the default connection factory on a particular WebLogic Server, you must select the Enable Default JMS Connection Factories checkbox when configuring the WebLogic Server.	None required. This is a configuration requirement. For more information, see "Server" in the Administration Console Online Help.
Connections	When closing a connection, the call blocks until outstanding synchronous calls and asynchronous listeners have completed.	None required.

Table 6-1 Existing Feature Functionality Changes (Continued)

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

Table 6-1 Existing Feature Functionality Changes (Continued)

Category	Description	Code Modification

Message Header Field	The `JMSMessageID` header field format has changed.	If you wish to access existing messages using the `JMSMessageID`, you may need to run one of the following `weblogic.jms.extensions.JMSHelper` methods to convert between WebLogic JMS 6.0 and pre-6.0 JMSMessageID formats.
		To convert from pre-6.0 to 6.0 `JMSMessageID` format:
		`public void oldJMSMessageIDToNew( String id, long timeStamp ) throws JMSException`
		To convert from 6.0 to pre- 6.0 `JMSMessageID` format:
		`public void newJMSMessageIDToOld( String id, long timeStamp ) throws JMSException`

Table 6-1 Existing Feature Functionality Changes (Continued)

Category	Description	Code Modification
Destinations	The `createQueue()` and `createTopic()` methods do not create destinations dynamically, only references to destinations that already exist given the vendor-specific destination name.	Update any portion of code that uses `createQueue()` or `createTopic()` to dynamically create destinations using the following JMSHelper class methods, respectively: `createPermanentQueueAsync()` and `createPermanentTopicAsync()`. For example, if your code used the following method to dynamically create a queue: `queue=qsession.createQueue(queueName);` You should modify the code to dynamically create a queue, as described in the sample `findQueue()` method in "Using the JMSHelper Class Methods" on page 4-42. For more information on the JMSHelper classes, see "Creating Destinations Dynamically" on page 4-42.
	When creating temporary destinations, you must specify a temporary template.	None required. This is a configuration requirement. For more information, see "JMS Templates" in the Administration Console Online Help.
	You must be the owner of the connection in order to create a message consumer for that temporary destination.	When creating a message consumer on a temporary destination, ensure that you are the owner of the connection.
Durable Subscribers	You no longer need to manually create JDBC tables for durable subscribers. They are created automatically.	None required.
	There is no limit on the number of durable subscribers that can be created.	None required.
	When defining a client ID programatically, it must be defined *immediately* after creating a connection. Otherwise, an exception will be thrown and you will be unable to make any other JMS calls on that connection.	Ensure that the `setClientID()` method is issued immediately after creating the connection. For more information, refer to "Defining the Client ID" on page 4-46.

Table 6-1 Existing Feature Functionality Changes (Continued)

Category	Description	Code Modification
Session Pools	Session pool factories, session pools, referenced connection factories, referenced destinations, and associated connection consumers must all be targeted on the same JMS server.	Ensure that all objects are targeted on the same JMS server.
	The `SessionPoolManager` and `ConnectionConsumerManager` interfaces that were published as part of the WebLogic JMS Version 5.1 javadoc have been removed from the Version 6.0 javadoc, as they are system interfaces and should not be used within client applications.	If used, remove any references to these objects from the client application.
	In WebLogic Server 6.0 SP2 or higher, for the `QueueConnection` and `TopicConnection` classes, the `MaxMessages` argument in the `createConnectionConsumer` method requires a specific value for the amount of messages to be reserved on the server. Therefore, `MaxMessages` will be parsed as follows: `-1` – The same as the default value, which is 10. `>0` – Positive integers require no conversion. `0` – An invalid value that will generate a `JMSException`. `<-1` – An invalid value that will generate a `JMSException`.	In the `createConnectionConsumer` method, ensure that the value of the `MaxMessages` argument is set to either -1 (the default) or a positive integer.
Transactions	To combine JMS and EJB database calls within the same transaction, a two-phase commit (2PC) license is required. In previous releases of WebLogic Server, it was possible to combine them by using the same database connection pool.	None required.
	Recovering or rolling back received queue messages makes them available to all consumers on the queue. In previous releases of WebLogic Server, rolled back messages were only available to the session that rolled back the message, until that session was closed.	None required.

Migrating Existing Applications

WebLogic Server 6.0 supports the JavaSoft JMS Specification version 1.0.2. and the latest JMS API – Errata. In order to use your existing JMS applications, you must first confirm your version of WebLogic server, and then perform the following migration procedure outlined in this section.

Before You Begin

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

- Version 4.5.1 — Migration is supported *only* for SP14. Customers running all service packs should contact BEA Support.
- Version 5.1 — Customers running with SP07 or SP08 should contact BEA Support before migrating existing JDBC stores to version 6.0.
  - In order to migrate object messages, the object classes need to be in the version 6.0 server classpath.

Migration Steps

Before you can use an existing JMS application, you must migrate the configuration and message data as follows:

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

Warning: Abruptly stopping the old version of WebLogic Server while messaging is still in process may cause problems during migration. Processing should be inactive before shutting down the old server and beginning the migration to WebLogic Server version 6.0.

2. Upgrade the WLS environment, as described in *Installing WebLogic Server*.

3. Migrate configuration information using the configuration conversion facility.

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

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

   The first two connection factories are deprecated, but they are still defined and usable for backwards compatibility. For information on the new default connection factory, see the table "Existing Feature Functionality Changes" on page 6-2.

   The JMS administrator will need to review the resulting configuration to ensure that the conversion meets the needs of the application.

   In this case, all of the JMS attributes will be mapped to a single node, as in Version 5.1.

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

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

   a. Make a backup of the existing JDBC database.

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

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

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

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

      Both the existing and new database information will exist on disk while the migration is performed, doubling the space requirements. Once migration is complete, you can delete the old JDBC database stores, as described in "Deleting JDBC Database Stores" on page 6-9.

5. Update any existing code, as required, to reflect the feature functionality changes described in "Existing Feature Functionality Changes" on page 6-1.

When you initially boot up the WebLogic Server, the existing JDBC database stores will be migrated automatically. If the automatic migration fails for any reason, the automatic migration will be re-attempted the next time the WebLogic Server boots.

Deleting JDBC Database Stores

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

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

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

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

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

CHAPTER

A Configuration Checklists

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

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

For more information on setting the configuration attributes, refer to the *Administration Guide*. For detailed descriptions of each of the configuration attributes, refer to the Administration Console Online Help.

Server Clusters

To support server clusters, configure the following:

- ☐ WebLogic Server targets under the Targets tab on the Connection Factories node
- ☐ WebLogic Server targets under the Targets tab on the JMS Servers node

JTA User Transactions

To support JTA user transactions, configure the following:

- ☐ Connection factory JTA user transaction mode by selecting the User Transactions Enabled checkbox under the Configuration—Transactions tab on the Connection Factories node

JMS Transactions

To support JMS transacted sessions, configure the following:

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

Message Delivery

To define message delivery attributes, configure the following:

- Connection factory priority, time-to-live, and delivery mode attributes under the Configuration—General tab on the Connection Factories node
- Destination priority, time-to-live, and delivery mode override attributes under the Configuration—Overrides tab on the Destinations node

Note: These settings can also be set dynamically by the message producer when sending a message or using the set methods, as described in "Sending Messages" on page 4-22.

The destination configuration attributes take precedence over all other settings.

Asynchronous Message Delivery

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

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

Persistent Messages

Note: Topic destinations are persistent if, and only if they have durable subscriptions. For more information about durable subscriptions, see "Setting Up Durable Subscriptions" on page 4-46.

To support persistent messaging, configure the following:

- ☐ Create a file or JDBC store using the Stores node
- ☐ JMS server backing store by setting the Store attribute under the Configuration—General tab on the JMS Servers node

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

- ☐ Default message delivery mode by setting one of the following attributes to `PERSISTENT` or `NON_PERSISTENT`:
  - Default Delivery Mode attribute under the Configurations—General tab on the Connection Factories node
  - Delivery Mode Override attribute under the Configurations—Overrides tab on the Destination node

  Note: You can also specify persistent as the delivery mode when sending messages, as described in "Sending Messages" on page 4-22.

Concurrent Message Processing

To support concurrent message processing, configure the following:

- ☐ Server session pool attributes under the Configuration tab on the Session Pools node
- ☐ Connection consumer attributes under the Configuration tab on the Connection Consumers node

Note: Server session pool factories, used for concurrent message processing, are not configurable. WebLogic JMS defines one ServerSessionPoolFactory object, by default: `weblogic.jms.ServerSessionPoolFactory:<name>`, where <*name*> specifies the name of the JMS server on which the session pool is created. For more information about using server session pool factories, refer to "Defining Server Session Pools" on page 4-61.

Multicasting

Note: Multicasting applies to topics only.

To configure multicasting on a topic, configure the following:

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

Durable Subscriptions

To support durable subscriptions, optionally configure the following:

- Client identifier (client ID) that can be used for clients with durable subscriptions by setting the ClientID attribute under the Configuration—General tab on the Connection Factories node

Note: Alternatively, clients can set the client ID in the connection after the connection is created, as described in "Setting Up Durable Subscriptions" on page 4-46.

Destination Sort Order

To support destination sort order, configure the following:

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

Temporary Destinations

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

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

Thresholds and Quotas

To configure thresholds and quotas, configure the following:

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

- Maximum number of messages that can be accumulated by a connection consumer by setting the Messages Maximum attribute under the Configuration tab of the Consumers node

B JDBC Database Utility

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

- Overview
- About JMS Stores
- Regenerating JDBC Stores

Overview

The JDBC `utils.Schema` utility allows you to regenerate new JDBC stores by deleting the existing versions. Running this utility is usually not necessary, since JMS automatically creates these stores for you. However, if your existing JDBC database stores somehow become corrupted, you can regenerate them using the `utils.Schema` utility.

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

About JMS Stores

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

- `<prefix>JMSStore`
- `<prefix>JMSState`

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

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

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

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

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

For instructions on creating and configuring a store, see "JMS File Stores" and "JMS JDBC Stores" for information about file and JDBC database stores, respectively, in the Administration Console Online Help.

Regenerating JDBC Stores

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

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

By convention, the DDL file has a `.ddl` extension. DDL files are provided for Cloudscape, Sybase, Oracle, MSSQL Server, and IBM DB2 databases.

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

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

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

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

Table 6-2 utils.Schema Command-Line Arguments

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

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

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

With the Cloudscape database, no username or password is required. However, the Cloudscape JDBC driver uses the `cloudscape.system.home` system property to find the directory containing its database files. You must supply the value for this property with the `-D` Java command option. In addition, you must specify the Cloudscape classes in your `CLASSPATH`, which exists in `weblogic/samples/eval/cloudscape/lib`.

For example, the following command creates the JMS tables in a Cloudscape server:

```
java
-Dcloudscape.system.home=/weblogic/samples/eval/cloudscape/data
  utils.Schema jdbc:cloudscape:demoPool;create=true
  COM.cloudscape.core.JDBCDriver -verbose
  weblogic/classes/jms/ddl/jms_cloudscape.ddl
```

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

Index

D

E

F

H

J

M

N

P

Q

R

S

T

U

X