BEA WebLogic Server™

Programming WebLogic JMS

Release 8.1
Document Date: December 2002
Revised: December 9, 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic JMS

Part Number	Date	Software Version
N/A	December 9, 2002	BEA WebLogic Server Version 8.1 Beta

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. Using WebLogic JMS with EJBs and Servlets

B. JDBC Database Utility

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, "Using WebLogic JMS with EJBs and Servlets," describes "best practice" methods that make it easier to use WebLogic JMS in conjunction with J2EE components, like Enterprise Java Beans and Servlets.
- Chapter 7, "WebLogic JMS Thin Client," describes how to access and deploy a small, yet full-featured version of WebLogic Server on the client-side for JMS applications.
- Chapter 8, "Porting WebLogic JMS Applications," describes how to port your WebLogic JMS applications to a new release of WebLogic Server.
- Appendix A, "Configuration Checklists," provides monitoring checklists for various WebLogic JMS features.

- Appendix B, "JDBC Database Utility," describes how to use the JDBC database utility to generate new JDBC stores and delete existing ones.

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 the 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://edocs.bea.com/wls/docs81b.

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 Specification and Javadoc supplied on Sun Microsystems' Java Web site at the follow location:

http://http://java.sun.com/products/jms/docs.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. *Example*s: 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.

CHAPTER

1 Introduction to WebLogic JMS

The following sections provide an overview of the Java Message Service (JMS) for BEA WebLogic Server:

- What Is JMS?
- Implementation of Java Specifications
- WebLogic JMS Features
- WebLogic JMS Architecture
- WebLogic JMS Extensions
- JMS Enhancements in WebLogic Server 8.1

What Is JMS?

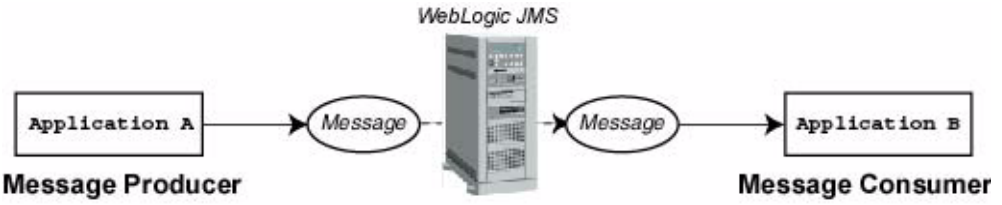
An enterprise messaging system, also referred to as Message-Oriented Middleware (MOM), enables applications to communicate with one another through the exchange of messages. A message is a request, report, and/or event that contains information needed to coordinate communication between different applications. 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.

Implementation of Java Specifications

WebLogic Server is compliant with the following Java specifications.

J2EE Specification

WebLogic Server 8.1 is compliant with Sun Microsystems' J2EE 1.3 specification.

JMS Specification

WebLogic Server 8.1 is fully compliant with the JMS Specification – version 1.0.2b and can be used in production.

WebLogic JMS Features

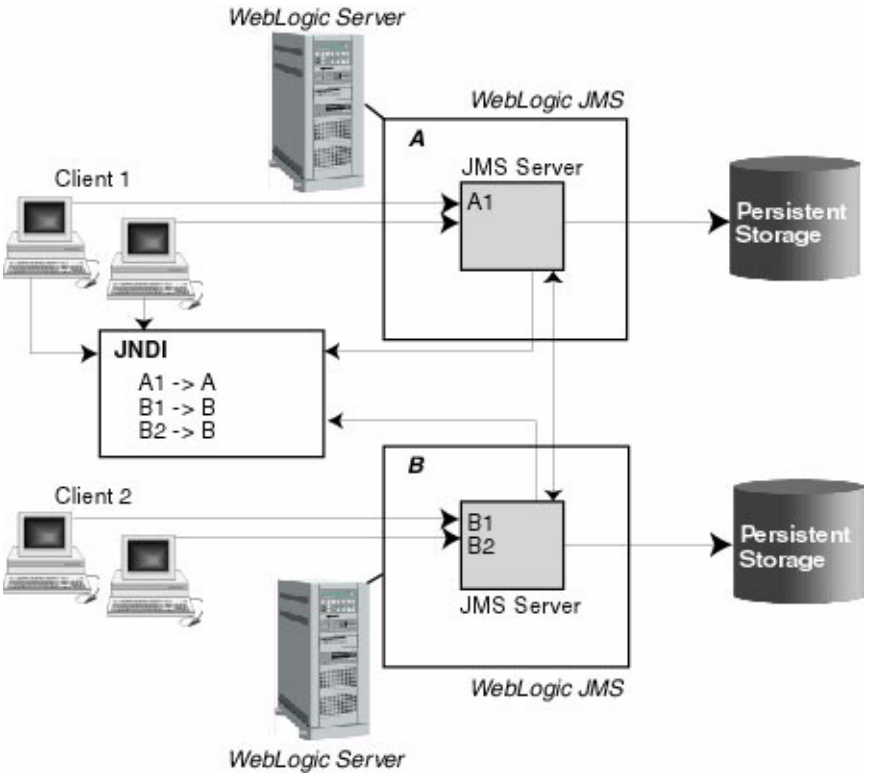
WebLogic JMS provides a full implementation of the JMS API. Specifically, WebLogic JMS:

- Provides a single, unified messaging API.
- Strictly adheres to the JMS Specification – version 1.0.2b.
- Supports clustering.
- Supports messaging for applications that span different operating systems and machine architectures.
- Can be configured by setting attributes from the WebLogic *Administration Console Online Help* 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, including support for distributed transactions and the two-phase commit protocol. JMS applications can also participate in transactions with other Java APIs that use JTA, including non-WebLogic XA compliant message brokers.
- 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.
- Can use either a database or a file for persistent message storage.
- Can be used with other 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

- Persistent storage (file or database) for storing persistent message data

Clustering Features

The WebLogic JMS architecture implements *clustering* of multiple JMS servers by supporting cluster-wide, transparent access to destinations from any server in the cluster. Although WebLogic Server supports distributing JMS destinations and connection factories throughout a cluster, JMS topics and queues are still managed by individual WebLogic Server instances in the cluster.

For more information about configuring clustering for WebLogic JMS, see "Configuring WebLogic JMS Clustering" on page 3-7. For detailed information about WebLogic Server clustering, see *Using WebLogic Server Clusters*.

The advantages of clustering include the following:

- *Load balancing of destinations across multiple servers in the cluster*
  - An administrator can establish load balancing of destinations across multiple 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.
  - An administrator can also configure multiple physical destinations as members of a single distributed destination set within a cluster. Producers and consumers are able to send and receive to the distributed destination. In the event of a single server failure within the cluster, WebLogic JMS then distributes the load across all available physical destination members within the distributed destination set.

    For more information on distributed destinations, see "Distributed Destination Tasks" in the *Administration Console Online Help*.
- *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.

- *Migratability*

  As an "exactly-once" service, WebLogic JMS takes advantage of the migration framework implemented in WebLogic Server for clustered environments. This allows WebLogic JMS to properly respond to migration requests and bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure. For more information, see "Configuring JMS Migratable Targets" on page 3-10.

Note: Automatic failover is not supported by WebLogic JMS for this release. For information about performing a manual failover, refer to "Recovering from a WebLogic Server Failure" on page 3-15.

WebLogic JMS Extensions

In addition to the API specified by Sun Microsystems' JMS Specification, WebLogic JMS provides a public API, `weblogic.jms.extensions`, which includes classes and methods for the extensions described in the following table.

Extension	For more information. . .
Create XML messages	Refer to "Step 6a: Create the Message Object (Message Producers)" on page 4-13
Define a session exception listener	Refer to "Defining a Session Exception Listener" on page 4-47
Set or display the maximum number of pre-fetched asynchronous messages allowed on the session	Refer to "Dynamically Configuring Multicasting Configuration Attributes" on page 4-87
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 4-87
Dynamically create permanent queues or topics	Refer to "Creating Destinations Dynamically" on page 4-49
Dynamically delete permanent queues or topics	Refer to "Deleting Destinations Dynamically" on page 4-52
Convert between WebLogic JMS 8.1 and pre-release 6.0 `JMSMessageID` formats	Refer to "Setting Message Header Fields" on page 4-63
Set a redelivery delay for messages	Refer to "Setting a Redelivery Delay for Messages" on page 4-34
Set a message delivery time for producers	Refer to "Setting a Delivery Time on Producers" on page 4-39
Set a delivery time for messages	Refer to "Setting a Delivery Time on Messages" on page 4-39
Set a scheduled delivery time for messages	Refer to "Setting a Scheduled Time-to-Deliver Override" on page 4-40

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.

JMS Enhancements in WebLogic Server 8.1

The following JMS enhancements are new to this release of WebLogic Server.

JMS Thin Client

At approximately 400k, the `wljmsclient.jar` file provides full WebLogic JMS functionality, yet greatly reduces the client-side WebLogic footprint by using a smaller library that contains only the set of supporting files required by client-side programs. The new client `.jar` file is available in the *`WL_HOME`*`/server/lib` subdirectory of the WebLogic Server installation directory (for example, `c:\bea\weblogic81b\server\lib`).

This `.jar` provides for full-featured WebLogic Server clients that can support clustering, load balancing, transactions, security, and failover. See "WebLogic JMS Thin Client" on page 7-1 for more information.

Accessing Foreign JMS Providers

Using the Foreign JMS Server node on the Administration Console, you can quickly map a foreign JMS provider so that its connection factories and destinations appear in the WebLogic JNDI tree as a local JMS objects. A Foreign JMS Server configuration

can also be used to reference remote instances of WebLogic Server in another cluster or domain in the local WebLogic JNDI tree. See "Accessing Foreign JMS Providers" in the *Administration Console Online Help* for more information.

Accessing JMS via Servlets and EJBs

New "wrappers" make it easier to use JMS inside a J2EE component. The wrappers provide features including automatic pooling of JMS Connection and Session objects (and some pooling of `MessageProducer` objects as well); automatic transaction enlistment for JMS providers that support XA; monitoring of the JMS connection and re-establishment after a failure; and security credentials that are managed by the container. See "Using WebLogic JMS with EJBs and Servlets" on page 6-1 for more information.

Better Expired Message Handling

Active message expiration ensures that expired messages are cleaned up immediately. Moreover, expired message auditing gives you the option of tracking expired messages, either by logging when a message expires or by redirecting expired messages to a special destination. See "Handling Expired Messages" in the *Administration Console Online Help* for more information.

Improved Message Flow Control by Blocking Producers

The "Blocking Send" features help you to avoid receiving message quota errors by temporarily blocking message producers from sending messages to a destination (queue or topic) when the destination has exceeded its specified maximum message quota. See "Avoiding Quota Exceptions by Blocking Message Producers" in the *Administration Console Online Help* for more information.

CHAPTER

2 WebLogic JMS Fundamentals

The following sections describe WebLogic JMS components and features:

Note: For more information on the JMS classes described in this section, access the JMS Specification and Javadoc supplied on the Sun Microsystems' Java web site at the following location:
http://java.sun.com/products/jms/docs.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.

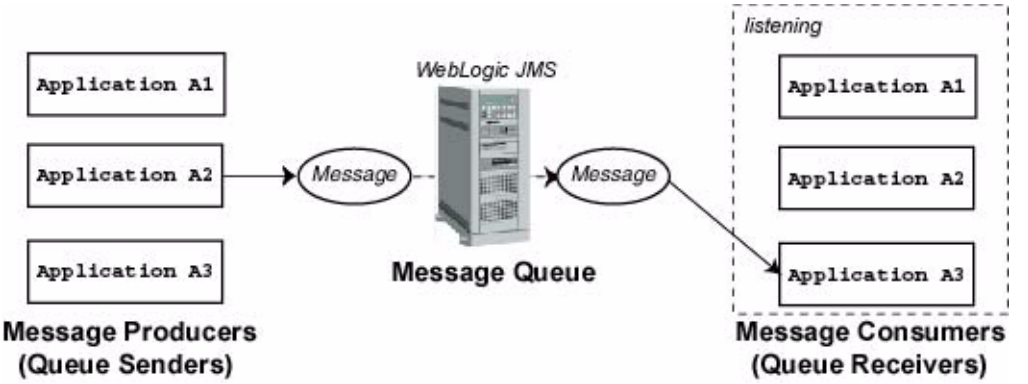
Note: The terms *producer* and *consumer* are used as generic descriptions of applications that send and receive messages, respectively, in either messaging model. For each specific messaging model, however, unique terms specific to that model are used when referring to producers and consumers.

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.

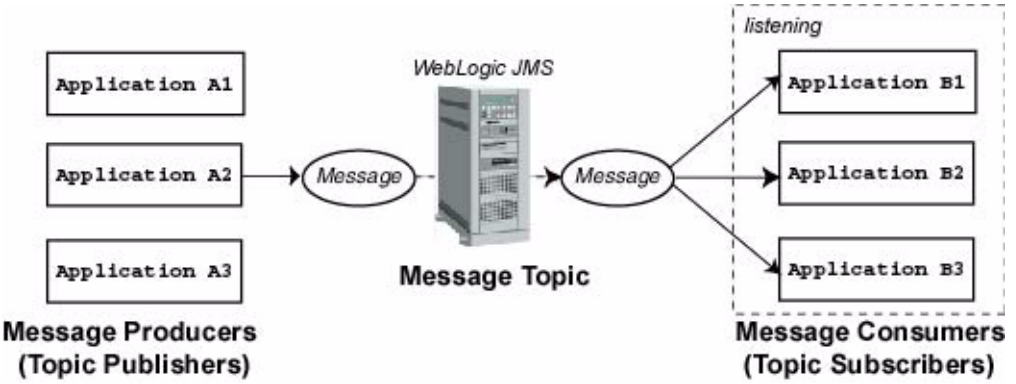
If multiple queue receivers are listening for messages on a queue, WebLogic JMS determines which one will receive the next message on a first come, first serve basis. If no queue receivers are listening on the queue, messages remain in the queue until a queue receiver attaches to the queue.

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

The Pub/sub messaging model supports durable subscribers, allowing you to assign a name to a topic subscriber and associate it with a user or application. For more information about durable subscribers, see "Setting Up Durable Subscriptions" on page 4-57.

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 WebLogic JMS Classes

JMS Class	Description
`ConnectionFactory`	Encapsulates connection configuration information. A connection factory is used to create connections. You look up a connection factory using JNDI.
`Connection`	Represents an open communication channel to the messaging system. A connection is used to create sessions.
`Session`	Defines a serial order for the messages produced and consumed.
`Destination`	Identifies a queue or topic, encapsulating the address of a specific provider. Queue and topic destinations manage the messages delivered from the PTP and Pub/sub messaging models, respectively.
`MessageProducer and MessageConsumer`	Provides the interface for sending and receiving messages. Message producers send messages to a queue or topic. Message consumers receive messages from a queue or topic.
`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 WebLogic JMS Classes

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 is 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.

The system administrator can also 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. For more information on JMS clustering, refer to "Configuring WebLogic JMS Clustering" on page 3-7.

WebLogic JMS defines one default connection factory. It can be looked up using the JNDI name, `weblogic.jms.ConnectionFactory`. 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 "Configuring JMS" in the *Administration Console Online Help*.

Notes: For backwards compatibility, WebLogic JMS still supports two deprecated default connection factories. The JNDI names for these factories are: `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnectionFactory`.

For information on migrating to a new default or user-defined connection factory from a deprecated connection factory, refer to "Porting WebLogic JMS Applications" on page 8-1.

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 4-46.

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.

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

Acknowledge Mode	Description
`CLIENT_ACKNOWLEDGE`	The `Session` object relies on the application to call an acknowledge method on a received message. Once the method is called, the session acknowledges all messages received since the last acknowledge. This mode allows an application to receive, process, and acknowledge a batch of messages with one call. **Note:** In the Administration Console, if the Acknowledge Policy attribute on the connection factory is set to `Previous`, but you want to acknowledge *all* received messages for a given session, then use the last message to invoke the acknowledge method. For more information on the Acknowledge Policy attribute, see "JMS Connection Factories" in the *Administration Console Online Help*.
`DUPS_OK_ACKNOWLEDGE`	The `Session` object acknowledges receipt of a message once the receiving application method has returned from processing it; duplicate acknowledges are permitted. This mode is most efficient in terms of resource usage. **Note:** You should avoid using this mode if your application cannot handle duplicate messages. Duplicate messages may be sent if an initial attempt to deliver a message fails.
`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.

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

Acknowledge Mode	Description
`MULTICAST_NO_ACKNOWLEDGE`	Multicast mode with no acknowledge required. Messages sent to a `MULTICAST_NO_ACKNOWLEDGE` session share the same characteristics as `NO_ACKNOWLEDGE` mode, described previously. This mode is supported for applications that want to support multicasting, and that do not require the quality of service provided by session acknowledge. For more information on multicasting, see "Using Multicasting" on page 4-83. **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 5-5.

Destination

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

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

Note: Administrators can also configure multiple physical destinations as members of a single distributed destination set within a server cluster. For more information, see "Distributed Destination" on page 2-13.

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 4-68.

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.

Distributed Destination

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

- For more information on using distributed destinations with your applications, see "Using Distributed Destinations" on page 4-90.
- For instructions on configuring distributed destinations using the Administration Console, see "Configuring Distributed Destinations" in the *Administration Console Online Help*.

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 4-57.

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 4-62, 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.	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
JMSDeliveryTime	Defines the earliest absolute time at which a message can be delivered to a consumer. This field can be used to sort messages in a destination and to select messages. For purposes of data type conversion, the JMSDeliveryTime is a long integer.	send() method
JMSDestination	Specifies the destination (queue or topic) to which the message is to be delivered. The application's message producer sets the value of this field when the message is sent. When a message is sent, this value is ignored. When a message is received, its destination value must be equivalent to the value assigned when it was sent.	send() method
JMSExpiration	Specifies the expiration, or time-to-live value, for a message. WebLogic JMS calculates the JMSExpiration value as the sum of the application's time-to-live and the current GMT. If the application specifies time-to-live as 0, JMSExpiration is set to 0, which means the message never expires. WebLogic JMS removes expired messages from the system to prevent their delivery.	send() method

Table 2-8 Message Header Fields (Continued)

Field	Description	Defined by
`JMSMessageID`	Contains a string value that uniquely identifies each message sent by a JMS Provider. All `JMSMessageID`s start with an `ID:` prefix. When a message is sent, this value is ignored. When the message is received, it contains a provider-assigned value.	`send()` method
`JMSPriority`	Specifies the priority level. This field is set before a message is sent. JMS defines ten priority levels, 0 to 9, 0 being the lowest priority. Levels 0-4 indicate gradations of *normal* priority, and level 5-9 indicate gradations of *expedited* priority. When the message is received, it contains the value specified by the method sending the message. You can sort destinations by priority by configuring a destination key, as described in "Configuing JMS" in the *Administration Console Online Help*.	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 4-31.	WebLogic JMS

Table 2-8 Message Header Fields (Continued)

Field	Description	Defined by
`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
`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 4-70.

For information about setting message property fields, see "Setting and Browsing Message Header and Property Fields" on page 4-62, 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 4-73, 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 4-73, 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 4-73, 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 4-73, or the `javax.jms.ConnectionConsumer` Javadoc.

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

CHAPTER

3 Managing WebLogic JMS

The WebLogic Server Administration Console provides an interface for easily enabling, configuring, and monitoring the features of the WebLogic Server, including JMS. To invoke the Administration Console, refer to the procedures described in "Starting and Stopping Servers" in the *Administration Console Online Help*.

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

Configuring WebLogic JMS

Using the Administration Console, you define configuration attributes to:

- Enable JMS.

- Create JMS servers and target a WebLogic Server instance or a Migratable Target where the JMS server will be deployed.
- Create and/or customize values for JMS servers, connection factories, destinations (physical queues and topics), distributed destinations (sets of physical queue and topic members within a cluster) destination templates, destination sort ordering (using destination keys), persistent stores, paging stores, session pools, and connection consumers.
- Set up custom JMS applications.
- Define thresholds and quotas.
- Enable any desired JMS features, such as:
  - server clustering using multiple connection factories
  - concurrent message processing via session pools
  - persistent messages and durable subscribers
  - paging out messages during peak loads to free up memory
  - controlling message flow during peak loads

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, WebLogic Server will not boot JMS when you restart it. A sample `examplesJMSServer` configuration is provided with the product in the Examples Server. For more information about starting the Examples Server, see "Starting the Default, Examples, and Pet Store Servers" in the *Installing WebLogic Server*.

When migrating from a previous release of Weblogic Server, the configuration information is converted automatically, as described in "Porting Existing Applications" on page 8-8.

To configure WebLogic JMS attributes, follow the procedures described in the "Configuring JMS" section of the *Administration Console Online Help*, to create and configure the JMS objects. Once WebLogic JMS is configured, applications can send and receive messages using the JMS API. For more information about developing WebLogic JMS applications, refer to "Developing a WebLogic JMS Application" on page 4-1.

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

Starting WebLogic Server and Configuring JMS

The following sections review how to start WebLogic Server and the Administration console, as well as provide a procedure for configuring a basic JMS implementation.

Starting the Default WebLogic Server

The default role for a WebLogic Server is the Administration Server. If a domain consists of only one WebLogic Server, that server is the Administration Server. If a domain consists of multiple WebLogic Servers, you must start the Administration Server first, and then you start the Managed Servers.

For complete information about starting the Administration Server, see "Starting and Stopping Servers" in the *Administration Console Online Help*.

Starting the Administration Console

The Administration Console is the Web-based administrator front-end (administrator client interface) to WebLogic Server. You must start the server before you can access the Administration Console for a server.

For complete details about using the Administration Console to configure a WebLogic Server, see "Starting and Using the Administration Console" in *Configuring and Managing a WebLogic Server*.

Configuring a Basic JMS Implementation

This section describes how to configure a basic JMS implementation using the Administration Console.

1. Under the Services node in the left pane, click the JMS node to expand the list.
2. Optionally, create a File Store for storing persistent messages in a flat file, and/or a Paging Store for swapping messages out to memory:

a. Click the Stores node in the left pane, and then click the Configure a new JMS File Store link in the right pane.

b. On the General tab, give the store a name, specify a directory, and then click the Create button.

c. Repeat these steps to create a Paging Store.

Note: For more information on configuring file stores, see "JMS File Store Tasks" in the *Administration Console Online Help*.

3. Optionally, create a JDBC Store for storing persistent messages in a database:

a. Click the JDBC node in the left pane to expand it.

b. Click the Connection Pools node in the left pane, and then click the Configure a new JDBC Connection Pool link in the right pane.

c. On the Configuration tabs, set the attributes for the connection pool, such as Name, URL, and database Properties. Click Apply on each tab when you're done making changes.

d. On the Target and Deploy tab, target an independent WebLogic Server instance or a server cluster on which to deploy the connection pool by selecting the appropriate check box, and then click Apply.

e. Return to the JMS –> Stores node, and then click the Configure a new JMS JDBC Store link in the right pane.

f. Give the JDBC Store a name, select a connection pool, and a prefix name. Then click Create.

Note: For more information on configuring JDBC-accessible JMS JDBC stores, see "JMS JDBC Store Tasks", "Configuring JDBC Connection Pools," "Configuring JDBC Multipools," and "Configuring JDBC DataSources" in the *Administration Console Online Help*.

4. Optionally, create a JMS Template to define multiple destinations with similar attribute settings. You also need a JMS Template to create temporary queues.

a. Click the Templates node in the left pane, and then click the Configure a new JMS Template link in the right pane.

b. On the General tab, give the template a name, and then click Create.

c. Fill in the Thresholds & Quotas, Override, Expiration Policy, and Redelivery tabs, as appropriate. Click Apply on each tab when you're done making changes.

Note: For more information on configuring a JMS Template, see "JMS Template Tasks" in the *Administration Console Online Help*.

5. Configure a JMS Server, as follows:

   a. Click the Server node in the left pane, and then click the Configure a new JMS Server link in the right pane.

   b. On the General tab, give the server a name, select a Store if you created one, select a Paging Store if you created one, and select a Template if you created one. Then click Create.

   c. Fill in the Thresholds & Quotas tab, as appropriate. Click Apply when you're done making changes.

   d. On the Target and Deploy tab, target an independent WebLogic Server instance or a Migratable Target server on which to deploy the JMS server by selecting the appropriate check box, and then click Apply.

   Note: For more information on configuring a JMS Server, see "JMS Server Tasks" in the *Administration Console Online Help*.

6. Create the JMS Destinations, which are queues (Point-To-Point) or topics (Pub/Sub):

   a. Under the Servers node in the left pane, click your new JMS server instance to expand the list, and then click the Destinations node.

   b. Click either the Configure a new JMS Queue or Configure a new JMS Topic link in the right pane.

   c. On the General tab, give the destination a name and a JNDI name. Fill in the other attributes, as appropriate, and then click Create.

   d. Fill in the Thresholds & Quotas, Override, Redelivery, Expiration Policy, and Multicast (for topics only) tabs, as appropriate. Click Apply on each tab when you're done making changes.

   Note: For more information on configuring a Destinations, see "JMS Destination Tasks" in the *Administration Console Online Help*.

7. Create a Connection Factory to enable your JMS clients to create JMS connections:

   a. Click to the expand the Connection Factory node and in the left pane, and then click the Configure a new JMS Connection Factory link in the right pane.

   b. On the General tab, give the connection factory a name and a JNDI name. Fill in the other attributes, as appropriate, and then click Create.

   c. Fill in the Transactions and Flow Control tabs, as appropriate. Click Apply on each tab when you're done making changes.

   d. On the Target and Deploy tab, target an independent WebLogic Server instance or a server cluster on which to deploy the connection factory by selecting selecting the appropriate check box, and then click Apply.

   Note: For more information on configuring a Connection Factory, see "JMS Connection Factory Tasks" in the *Administration Console Online Help*.

8. Optionally, use the Destination Keys node to define the sort order of messages that arrive on a specific destination. For more information, see "JMS Destination Key Tasks" in the *Administration Console Online Help*.

9. Optionally, use the Distributed Destinations node to make your physical destinations part of a single distributed destination set within a server cluster. For more information, see "Distributed Destinations Tasks" in the *Administration Console Online Help*.

10. Optionally, create JMS Session Pools, which enable your applications to process messages concurrently, and Connection Consumers (queues or topics) that retrieve server sessions and process messages. For more information, see "JMS Session Pools Tasks" and "JMS Connection Consumers Tasks" in the *Administration Console Online Help*.

Configuring WebLogic JMS Clustering

A WebLogic Server *cluster* is a group of servers that work together to provide a more scalable, more reliable application platform than a single server. A cluster appears to its clients as a single server but is in fact a group of servers acting as one. A cluster provides three key features above a single server:

- Scalability—servers can be added to the cluster dynamically to increase capacity.
- High Availability—redundancy of multiple servers insulates applications from failures. Redundancy of multiple destinations (queues and topics) as members of a single *distributed destination* set within a cluster ensures redistribution of the messaging load to other available members in the set when one member becomes unavailable.
- Migratability—respond to migration requests and bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure.

A *clustered service* is an API or interface that is available on multiple servers in the cluster.

Note: JMS clients depend on unique WebLogic Server names to successfully access a cluster—even when WebLogic Servers reside in different domains. Therefore, make sure that *all* WebLogic Servers that JMS clients contact have unique server names.

For more information about starting WebLogic clusters and its features and benefits, see "Configuring WebLogic Servers and Clusters" in *Using WebLogic Server Clusters*.

How JMS Clustering Works

You 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 administrator can configure multiple JMS servers on the various nodes in the cluster—as long as the JMS servers are uniquely named—and can then assign JMS destinations to the various JMS 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 WebLogic Server.

JMS Clustering Requirements

The following guidelines apply when configuring WebLogic JMS to work in a clustered environment in a single WebLogic domain or in a multi-domain environment.

- All WebLogic Servers that JMS clients contact must have unique server names.
- All JMS servers targeted to nodes in the cluster must be uniquely named.
- If persistent messaging is required, all JMS stores must be uniquely named.

JMS Distributed Destination within a Cluster

The WebLogic JMS administrator can also configure multiple destinations as part of a single distributed destination set within a cluster. Producers and consumers are able to send and receive to the distributed destination. In the event of a single server failure within the cluster, WebLogic JMS then distributes the load across all available physical destinations within the distributed destination set. For more information, see "Distributed Destination Tasks" in the *Administration Console Online Help*.

JMS as a Migratable Service within a Cluster

WebLogic JMS takes advantage of the migration framework implemented in the WebLogic Server core for clustered environments. This allows WebLogic JMS to properly respond to migration requests and bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure. For more information, see "Configuring JMS Migratable Targets" on page 3-10.

Configuration Steps

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

1. Administer WebLogic clusters as described in "Configuring WebLogic Servers and Clusters" in *Using WebLogic Server Clusters*.
2. Identify *server targets* for JMS servers and for connection factories using the Administration Console:
   - For JMS servers, you can identify either a single-server target or a migratable target, which is a set of WebLogic Server instances in a cluster that can host an "exactly-once" service like JMS in case of a single server failure. For more information on migratable targets, see "Configuring JMS Migratable Targets" on page 3-10.
   - For connection factories, you can identify either a single-server target or a cluster target, which are WebLogic Server instances that are associated with a connection factory to support clustering.

   For more information about these configuration attributes, see "JMS Server Tasks" or "JMS Connection Factory Tasks" in the *Administration Console Online Help*.

   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.
3. Optionally, you can configure your physical destinations as part of a single distributed destination set within a cluster. For more information, see "Distributed Destination Tasks" in the *Administration Console Online Help*.

What About Failover?

For WebLogic JMS implementations that are part of a WebLogic 7.0 clustered environment, JMS offers service continuity in the event of a single Weblogic Server failure by enabling you to configure multiple physical destinations (queues and topics) as part of a single distributed destination set. In addition, implementing the Migratable Service feature, will ensure that pinned "exactly-once" services, like JMS, do not introduce a single point of failure for dependent applications in the cluster,

However, automatic failover is not currently supported by WebLogic JMS. For information about performing a manual failover, refer to "Recovering from a WebLogic Server Failure" on page 3-15.

Configuring JMS Migratable Targets

As an "exactly-once" service, WebLogic JMS is not active on all WebLogic Server instances in a cluster. It is instead "pinned" to a single server in the cluster to preserve data consistency. To ensure that pinned services do not introduce a single point of failure for dependent applications in the cluster, WebLogic Server can be configured to migrate exactly-once services to any server in the migratable target list.

WebLogic JMS takes advantage of the migration framework by allowing an administrator to specify a migratable target for a JMS server in the Administration Console. Once properly configured, a JMS server and all of its destinations can migrate to another WebLogic Server within a cluster.

This allows WebLogic JMS to properly respond to migration requests and bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure with the cluster.

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

How JMS Migration Works

For implementations that are part of a WebLogic clustered environment, WebLogic JMS implements the `weblogic.cluster.Migratable` interface, which allows JMS servers to respond to activate and deactivate requests.

Table 3-1 WebLogic JMS Migration Process

Migration state...	What takes place...
Initialization	Initialization of a JMS server includes processing any configuration or deployment information and creating the appropriate objects. Destinations and other JMS resources are unavailable at this time. In addition, the persistent store is not opened, as this could compromise the integrity of the store. The JMS server makes itself available to handle changes in configuration that may occur between initialization and activation.
Activation	When a JMS server is activated, it opens the persistent store, performs any necessary recovery, reconciles the contents of the store with the current configuration, and makes the destinations available for access by applications. In addition, any configured server session pools begin processing after activation is complete.
Deactivation	When a JMS server is deactivated it stops all server session pool processing, marks all destinations as unavailable, flushes and closes its persistent stores, purges its destinations, and deletes all objects for the JMS server.

Configuration Steps

In order to make WebLogic JMS a migratable service in a clustered environment, you must do the following:

1. Administer WebLogic clusters as described in "Configuring WebLogic Servers and Clusters" in the *Using WebLogic Server Clusters*.
2. Configure a migratable target for the cluster as described in "Server -> Control -> JMS Migration Config. -> " in the *Administration Console Online Help*.

3. Identify a migratable target server on which to deploy a JMS server as described in "JMS Server Tasks" in the *Administration Console Online Help*.

   When a migratable target server boots, the JMS server boots as well on the user-preferred server in the cluster. However, a JMS server and all of its destinations can migrate to another server within the cluster in response to a WebLogic Server failure or due to a scheduled migration for maintenance.

   Note: A JMS server and all of its destination members can migrate to another WebLogic Server within a cluster—even when the target WebLogic Server is already hosting a JMS server with all of its destination members. Although this can lead to situations where the same WebLogic server hosts two physical destinations for a single distributed destination, this is permissible in the short term, since the WebLogic Server can host multiple physical destinations for that distributed destination. For more information about JMS distributed destinations, see "Using Distributed Destinations" on page 4-90.

4. For implementations that use persistent messaging, make sure that the persistent store is configured such that all the candidate servers in a migratable target share access to a persistent store. For more information about migrating persistent stores, see "Persistent Store Migration" on page 3-12.

5. The administrator can manually migrate services before performing server maintenance or to a healthy server if the host server fails.

Persistent Store Migration

Weblogic JMS persistent stores cannot be migrated along with JMS servers; therefore, applications that need access to persistent stores from other physical machines after the migration of a JMS server must implement an alternative solution, as follows:

- Implement a hardware solution, such as a dual-ported SCSI disk or Storage Area Network (SAN) to make your JMS persistent store available from other machines.
- Use JDBC to access your JMS JDBC store, which can be on yet another server. Applications can then take advantage of any high-availability or failover solutions offered by your database vendor.

  For more information about configuring a JMS JDBC store, see "Configuring JDBC Stores" in the *Administration Console Online Help*.

Migration Failover

For information about procedures for recovering from a WebLogic Server failure, see "Recovering from a WebLogic Server Failure" on page 3-15.

Tuning WebLogic JMS

The following sections explain how to get the most out of your applications by implementing the administrative performance tuning features available with WebLogic JMS.

- Synchronous Write Policies for JMS File Stores—disabling synchronous writes improves file store performance, often quite dramatically, but at the expense of possibly losing sent messages or generating duplicate received messages in the event of an operating system crash or a hardware failure.

  For more information, see "Configuring a Synchronous Write Policy for JMS File Stores" in the *Administration Console Online Help*.

- Using Message Paging—you can free up valuable virtual memory during peak message load periods by swapping out messages from memory to persistent storage whenever your message loads reach a specified threshold. From a performance perspective, this feature can greatly benefit WebLogic Server implementations with the large message spaces that are required by today's enterprise applications.

  For more information, see "Using Message Paging" in the *Administration Console Online Help*.

- Establishing Message Flow Control—a JMS server or JMS destination (queue or topic) can be configured to instruct message producers to limit their message flow when it determines that it is becoming overloaded.

  For more information, see "Establishing Message Flow Control" in the *Administration Console Online Help*.

- Avoiding Quota Exceptions by Block Message Producers—the "Blocking Send" features help you to avoid receiving message quota errors by temporarily blocking message producers from sending messages to a destination (queue or topic) when it has exceeded its specified maximum message quota.

For more information, see "Avoiding Quota Exceptions by Blocking Message Producers" in the *Administration Console Online Help*.

- Active Expired Message Handling—active message expiration ensures that expired messages are cleaned up immediately. Moreover, expired message auditing gives you the option of tracking expired messages, either by logging when a message expires or by redirecting expired messages to a special destination.

  For more information, see "Handling Expired Messages" in the *Administration Console Online Help*.

- Tuning Distributed Destinations—the following attributes on the JMS Connection Factory can be configured to tune your distributed destinations:

  - Load Balancing—defines whether WebLogic JMS will spread or balance the messaging load across distributed destinations.
  - Server Affinity—defines whether a WebLogic Server that is attempting to load balance consumers or producers across multiple physical destinations in a distributed destination set, will first attempt to load balance across those physical destinations being served by any JMS servers that are also running on the same WebLogic Server.

  For more information, see "Tuning Distributed Destinations" in the *Administration Console Online Help*.

Monitoring WebLogic JMS

Statistics are provided for the following JMS objects: JMS servers, connections, sessions, destinations, durable subscribers, 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 "Monitoring JMS" in the *Administration Console Online Help.*

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

Programming Considerations

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

If a WebLogic Server Instance Fails and...	Then...
You are connected to the failed WebLogic Server instance	A `JMSException` is delivered to the connection exception listener. You must restart the application once the server is restarted or replaced.
You are not connected to the failed WebLogic Server instance	You must re-establish everything once the server is restarted or replaced.
A JMS Server is targeted on the failed WebLogic Server instance	A `ConsumerClosedException` is delivered to the session exception listener. You must re-establish any message consumers that have been lost.

Migrating JMS Data to a New Server

WebLogic JMS uses the migration framework implemented in the WebLogic Server core, which allows WebLogic JMS to properly respond to migration requests and bring a WebLogic JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure.

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

You can recover JMS data from a failed WebLogic Server by starting a new server and doing one or more of the tasks in Table 3-2.

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

Table 3-2 Migration Task Guide

If your JMS application uses. . .	Perform the following task. . .
Persistent messaging—JDBC Store	■ If the JDBC database store physically exists on the failed server, migrate the database to a new server and ensure that the JDBC connection pool URL attribute reflects the appropriate location reference. ■ If the JDBC database does not physically exist on the failed server, access to the database has not been impacted, and no changes are required.
Persistent messaging—File Store	Migrate the file to the new server, ensuring that the pathname within the WebLogic Server home directory is the same as it was on the original server.

If your JMS application uses. . .	Perform the following task. . .
Transactions	Migrate the transaction log to the new server by copying all files named `<servername>*.tlog`. This can be accomplished by storing the transaction log files on a dual-ported disk that can be mounted on either machine, or by manually copying the files. If the files are located in a different directory on the new server, update that server's `TransactionLogFilePrefix` server configuration attribute before starting the new server. **Note:** If migrating following a system crash, it is very important that the transaction log files be available when the server is restarted at its new location. Otherwise, transactions in the process of being committed at the time of the crash might not be resolved correctly, resulting in data inconsistencies. All uncommitted transactions are rolled back.

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

For information about starting a new WebLogic Server, refer to see "Starting and Stopping Servers" in the *Administration Console Online Help*. For information about recovering a failed server, refer to Recovering Failed Servers in the *Configuring and Managing WebLogic Domains* guide.

For more information about migratable targets, see "Configuring WebLogic Migratable Services" in *Using WebLogic Server Clusters.*

CHAPTER

4 Developing a WebLogic JMS Application

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

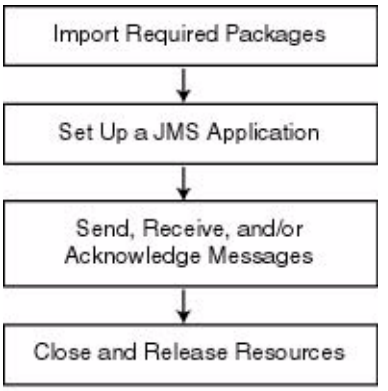
- "Filtering Messages" on page 4-70
- "Defining Server Session Pools" on page 4-73
- "Using Multicasting" on page 4-83
- "Using Distributed Destinations" on page 4-90

Note: For more information about the JMS classes described in this section, access the JMS Javadoc supplied on the Sun Microsystems' Java web site at the following location: http://java.sun.com/products/jms/docs.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	Sun Microsystems' 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;
```

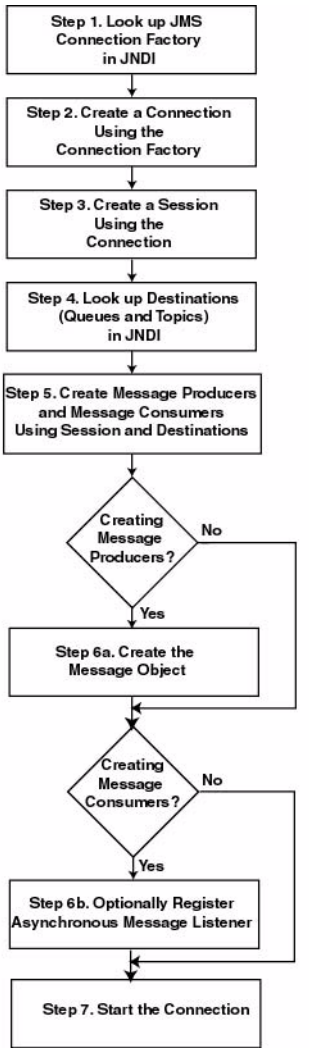
If you want to utilize any of the WebLogic JMS extension classes described in the previous table, also include the following statement on your import list:

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

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

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

For information about setting up transacted applications and JTA user transactions, see "Using Transactions with WebLogic JMS" on page 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 "Configuring JMS" in the *Administration Console Online Help*.

Once the connection factory has been defined, you can look it up by first establishing a JNDI context (`context`) using the `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 the `javax.jms.ConnectionFactory` Javadoc.

Step 2: Create a Connection Using the Connection Factory

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

For more information about the `Connection` class, see "Connection" 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 Table 2-5, "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 "Configuring JMS" in the *Administration Console Online Help*.

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 4-49.

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 4-49.

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 2-12 or the `javax.jms.Destination` Javadoc.

Server Affinity When Looking Up Destinations

The `createTopic()` and `createQueue()` methods also allow a "`JMS_Server_Name./Destination_Name`" syntax to indicate server affinity when looking up destinations. This way when a destination is locally deployed in the same JVM as the connection factory, the connection factory will only return names matching local destinations. If the name is not on the local JVM an exception is thrown, even though the same name might be deployed on a different JVM.

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

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

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

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

For more information about the `MessageProducer` and `MessageConsumer` classes, see "MessageProducer and MessageConsumer" on page 2-13, 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 4-70.

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 4-57. 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 4-70.

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 2-15, 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 check box when configuring the connection factory. For more information on configuring JMS, see "Configuring JMS" in the *Administration Console Online Help*.

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 4-47.

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 4-47.

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 2-13 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 4-46 or the `javax.jms.Connection` Javadoc.

Example: Setting Up a PTP Application

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server in the *`WL_HOME`*`\samples\server\src\examples\jms\queue` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. The `init()` method shows how to set up and start a `QueueSession` for a JMS application. The following shows the `init()` method, with comments describing each setup step.

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

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

private QueueConnectionFactory qconFactory;
```

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

Set up the JNDI initial context, as follows:

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

Note: When setting up the JNDI initial context for a servlet, use the following method:

```
Context ctx = new InitialContext();
```

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

```
public void init(
  Context ctx,
  String 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 *`WL_HOME`*`\samples\server\src\examples\jms\topic` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. The `init()` method shows how to set up and start a topic session for a JMS application. The following shows the `init()` method, with comments describing each setup step.

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

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

protected TopicConnectionFactory tconFactory;
protected TopicConnection tcon;
```

```
protected TopicSession tsession;
protected TopicPublisher tpublisher;
protected Topic topic;
protected TextMessage msg;
```

Set up the JNDI initial context, as follows:

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

Note: When setting up the JNDI initial context for a servlet, use the following method:

```
Context ctx = new InitialContext();
```

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

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

Step 1 Look up a connection factory using JNDI.

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

Step 2 Create a connection using the connection factory.

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

Step 3 Create a session using the connection. The following defines the session as non-transacted and specifies that messages will be acknowledged automatically. For more information about setting session transaction and acknowledge modes, see "Session" on page 2-8.

```
    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 4-13.

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 4-13. 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 `clearbody()` method to clear the existing contents and enable write permission.

For more information about the `MessageProducer` class, see "MessageProducer and MessageConsumer" on page 2-13 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 "Configuring JMS" in the *Administration Console Online Help*.
- Values specified using the message producer's set methods, as described in "Dynamically Configuring Message Producer Configuration Attributes" on page 4-26.

Note: WebLogic JMS also provides a proprietary `TimeToDeliver` attribute (that is, birth time), as described in "Dynamically Configuring Message Producer Configuration Attributes" on page 4-26.

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

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

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

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 "Configuring JMS" in the *Administration Console Online Help*.
- Values specified using the message producer's set methods, as described in "Dynamically Configuring Message Producer Configuration Attributes" on page 4-26.

Note: WebLogic JMS also provides a proprietary `TimeToDeliver` attribute (that is, birth time), as described in "Dynamically Configuring Message Producer Configuration Attributes" on page 4-26.

If you define the delivery mode as `PERSISTENT`, you should configure a backing store, as described in "Configuring JMS" in the *Administration Console Online Help*.

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

If the topic publisher is an anonymous producer (that is, if when the topic was created, the name was set to null), then you must specify the topic name (using either of the last two methods) to indicate where to deliver messages. For more information about defining anonymous producers, see "Create TopicPublishers and TopicSubscribers" on page 4-12.

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, time-to-live, and time-to-deliver values. If not specified, the delivery mode, priority, time-to-live, and time-to-deliver attributes are

set to the connection factory or destination override configuration attributes defined for the producer, as described "Configuring JMS" in the *Administration Console Online Help*.

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, time-to-live, time-to-deliver attribute settings can be overridden by the destination using the Delivery Mode Override, Priority Override, Time To Live Override, and Time To Deliver Override destination configuration attributes, as described in "Configuring Destinations" in the *Administration Console Online Help*.

Table 4-2 Message Producer Set and Get Methods

Attribute	Set Method	Get Method
Delivery Mode	`public void setDeliveryMode( int deliveryMode ) throws JMSException`	`public int getDeliveryMode( ) throws JMSException`
Priority	`public void setPriority( int defaultPriority ) throws JMSException`	`public int getPriority( ) throws JMSException`
Time-To-Live	`public void setTimeToLive( long timeToLive ) throws JMSException`	`public long getTimeToLive( ) throws JMSException`
Time-To-Deliver	`public void setTimeToDeliver( long timeToDeliver ) throws JMSException`	`public long getTimeToDeliver( ) throws JMSException`

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 *`WL_HOME`*`\samples\server\src\examples\jms\queue` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. The example shows the code required to create a `TextMessage`, set the text of the message, and send the message to a queue.

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

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

Example: Sending Messages Within a Pub/Sub Application

The following example is excerpted from the `examples.jms.topic.TopicSend` example, provided with WebLogic Server in the *`WL_HOME`*`\samples\server\src\examples\jms\topic` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. It shows the code required to create a `TextMessage`, set the text of the message, and send the message to a topic.

```
msg = tsession.createTextMessage();
       .
       .
       .
public void send(
  String message
) throws JMSException
```

```
{
  msg.setText(message);
  tpublisher.publish(msg);
}
```

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

Receiving Messages

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

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 "Configuring JMS" in the *Administration Console Online Help*, 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 "Configuring JMS" in the *Administration Console Online Help*.

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

For more information about the JMS classes for receiving messages and the message types, see the `javax.jms.Message` Javadoc. For information about sending messages, see "Sending Messages" on page 4-22.

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 *`WL_HOME`*`\samples\server\src\examples\jms\queue` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. Rather than set a message listener, you would call `qreceiver.receive()` for each message. For example:

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

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

Example: Receiving Messages Synchronously Within a Pub/Sub Application

The following example is excerpted from the `examples.jms.topic.TopicReceive` example, provided with WebLogic Server in the *`WL_HOME`*`\samples\server\src\examples\jms\topic` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. Rather than set a message listener, you would call `tsubscriber.receive()` for each message.

For example:

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

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

Recovering Received Messages

Note: This section applies only to non-transacted sessions for which the acknowledge mode is set to `CLIENT_ACKNOWLEDGE`, as described in Table 2-5, "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 Table 2-5, "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 4-46 or "Closing a Session" on page 4-48, respectively.

The following example is excerpted from the `examples.jms.queue.QueueSend` example, provided with WebLogic Server in the *`WL_HOME`*`\samples\server\src\examples\jms\queue` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation. This example shows the code required to close the message consumer, session, and connection objects.

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

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

Managing Rolled Back or Recovered Messages

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

- Setting a Redelivery Delay for Messages
- Setting a Redelivery Limit for Messages

Setting a Redelivery Delay for Messages

You can delay the redelivery of messages when a temporary, external condition prevents an application from properly handling a message. This allows an application to temporarily inhibit the receipt of "poison" messages that it cannot currently handle. When a message is rolled back or recovered, the redelivery delay is the amount of time a message is put aside before an attempt is made to redeliver the message.

If JMS immediately redelivers the message, the error condition may not be resolved and the application may still not be able to handle the message. However, if an application is configured for a redelivery delay, then when it rolls back or recovers a message, the message is set aside until the redelivery delay has passed, at which point the messages are made available for redelivery—as long as the error condition has already been resolved.

All messages consumed and subsequently rolled back or recovered by a session receive the redelivery delay for that session at the time of rollback or recovery. Messages consumed by multiple sessions as part of a single user transaction will receive different redelivery delays as a function of the session that consumed the individual messages. Messages that are left unacknowledged or uncommitted by a client, either intentionally or as a result of a failure, are not assigned a redelivery delay.

Setting a Redelivery Delay

A session inherits the redelivery delay from its connection factory when the session is created. The `RedeliveryDelay` attribute of a connection factory is configured using the Administration Console. For more information, see "JMS Connection Factory Tasks" in the *Administration Console Online Help*.

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

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

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

public long getRedeliveryDelay(
) throws JMSException;
```

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

Overriding the Redelivery Delay on a Destination

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

The `RedeliveryDelayOverride` attribute of a destination is configured using the Administration Console. For more information, see "JMS Destination Tasks" in the *Administration Console Online Help*.

Setting a Redelivery Limit for Messages

You can specify a limit on the number of times that WebLogic JMS will attempt to redeliver a message to an application. Once WebLogic JMS fails to redeliver a message to a destination for a specific number of times, the message can be redirected to an error destination that is associated to the message destination. If no error destination is configured, then the message is silently deleted.

Configuring a Message Redelivery Limit

When a destination's attempts to redeliver a message to a consumer reaches a specified redelivery limit, then the destination deems the message undeliverable. The `RedeliveryLimit` attribute is set on a destination and is configurable using the Administration Console. For more information, see " JMS Destination Tasks" in the *Administration Console Online Help*.

Configuring an Error Destination for Undelivered Messages

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

The `ErrorDestination` attribute is configured using the Administration Console. For more information, see " JMS Destination Tasks" in the *Administration Console Online Help*.

If a message redelivery attempt has already reached its specified redelivery limit, but its error destination has also reached its maximum quota, then the message is deemed undeliverable and is dropped. Non-persistent messages are deleted, while persistent messages remain in the store and will reappear in their originating destination (not the error destination) when the server is restarted. In either case, a log message is generated. To prevent the log file from becoming clogged, the log message is only generated once per error destination every five minutes, until the error condition is resolved.

Ordered Redelivery of Rolled Back Messages

All messages delivered to a consumer from a given producer are guaranteed to arrive at the consumer in the order in which they were produced. In addition, the "Ordered Redelivery" feature guarantees ordering of *redelivered* messages given certain constraints, like using message selection, having a sorted destination, delays in the rolling back of messages, and message consumption by other consumers.

Single Consumer

Ordered redelivery is only guaranteed when there is a single consumer. If there are multiple consumers, then there are no guarantees about the order in which any individual consumer will receive messages.

Note: With respect to MDBs (message-driven beans), the number of consumers is a function of the number of MDB instances deployed for a given MDB. The initial and maximum values for the number of instances must be "1". Otherwise no ordering guarantees can be made with respect to redelivered messages.

Sort Order

If a given destination is sorted, has JMS destination keys defined, and another message is produced such that the message would be placed at the top of the ordering, then no guarantee can be made between the redelivery of an existing message and the delivery of the incoming message.

Selection

If a consumer is using a selector, then ordering on redelivery is only guaranteed between the message being redelivered and other messages that match the criteria for that selector. There are no guarantees of order with respect to messages that do not match the selector.

Message Pipeline Size

For JMS applications using JTA transactions in conjunction with an asynchronous consumer, the size of the message pipeline must be "1". Anything value higher than "1" means there may be additional in-flight messages that may appear ahead of a redelivered message. The size of the pipeline can be set using the Messages Maximum attribute on the JMS connection factory used by the sending application. MDB applications must define an application specific connection factory, set the Messages Maximum attribute value to "1" on that connection factory, and then reference the connection factory in the EJB descriptor for their MDB application.

Note: The only application capable of using JTA transactions in conjunction with an asynchronous consumer are MDBs, and the WebLogic Messaging Bridge.

Performance Requirements

JMS applications that take advantage of this feature will incur performance degradation for asynchronous consumers using JTA transactions (specifically, MDBs and the WebLogic Message Bridge). This is caused by a mandatory reduction in the number of in-flight messages to exactly "1". This means no messages are aggregated when sent to the client.

Setting Message Delivery Times

You can schedule message deliveries to an application for specific times in the future. Message deliveries can be deferred for short periods of time (such as seconds or minutes) or for long stretches of time (for example, hours later for batch processing). You can also specify a relative delivery time (in milliseconds), which will then be computed into an absolute delivery time for a message. Until that delivery time, the message is essentially invisible until it is delivered, allowing you to schedule work at a particular time in the future.

Messages are not sent on a recurring basis; they are sent only once. In order to send messages on a recurring basis, a received scheduled message must be sent back to its original destination. Typically, the receive, the send, and any associated work should be under the same transaction to ensure exactly-once semantics.

Setting a Delivery Time on Producers

Support for setting and getting a time-to-deliver on an individual producer is provided through the `weblogic.jms.extensions.WLMessageProducer` interface, which is an extension to the `javax.jms.MessageProducer` interface. To define a time-to-deliver on an individual producer, use the following methods:

```
public void setTimeToDeliver(
  long timeToDeliver
) throws JMSException;

public long getTimeToDeliver(
) throws JMSException;
```

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

Setting a Delivery Time on Messages

Note: The message methods described here are similar to other JMS message methods that are set via the producer. Specifically, the setting of the delivery time is reserved for JMS providers. An application can set the value on a message, but the producer will override it when the message is sent or published.

The `DeliveryTime` is a JMS message header field that defines the earliest absolute time at which the message can be delivered. That is, the message is held by the messaging system and is not given to any consumers until that time.

As a JMS header field, the `DeliveryTime` can be used to sort messages in a destination or to select messages. For purposes of data type conversion, the delivery time is stored as a long integer.

The support for setting and getting the delivery time on a message is provided through the `weblogic.jms.extensions.WLMessage` interface, which is an extension to the `javax.jms.Message` interface. To define a delivery time on a message, use the following methods:

```
public void setJMSDeliveryTime(
  long deliveryTime
) throws JMSException;
```

```
public long getJMSDeliveryTime(
) throws JMSException;
```

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

Overriding a Delivery Time

When a producer is created it inherits its `TimeToDeliver` attribute, expressed in milliseconds, from the connection factory used to create the connection that the producer is a part of. Regardless of what time-to-deliver is set on the producer, the destination to which a message is being sent or published can override the setting. An administrator can set the `TimeToDeliverOverride` attribute on a destination in either a relative or scheduled string format.

Setting a Relative Time-to-Deliver Override

A relative `TimeToDeliverOverride` is a String specified as an integer, and is configurable using the Administration Console. For more information, see" JMS Destination Tasks" in the *Administration Console Online Help*.

Setting a Scheduled Time-to-Deliver Override

A scheduled `TimeToDeliverOverride` can also be specified using the `weblogic.jms.extensions.schedule` class, which provides methods that take a schedule and return the next scheduled time for delivering messages. A cron-like string is used to define the schedule. The format is defined by the following BNF syntax:

```
schedule := millisecond second minute hour dayOfMonth month
            dayOfWeek
```

The BNF syntax for specifying the `second` field is as follows:

```
second      := * | secondList
secondList  := secondItem [, secondList]
secondItem  := secondValue | secondRange
SecondRange := secondValue - secondValue
```

Similar BNF statements for milliseconds, minute, hour, day-of-month, month, and day-of-week can be derived from the second syntax. The values for each field are defined as non-negative integers in the following ranges:

```
milliSecondValue := 0-999
milliSecondValue := 0-999
secondValue      := 0-59
minuteValue      := 0-59
hourValue        := 0-23
dayOfMonthValue  := 1-31
monthValue       := 1-12
dayOfWeekValue   := 1-7
```

Note: These values equate to the same ranges that the `java.util.Calendar` class uses, except for `monthValue`. The `java.util.Calendar` range for `monthValue` is 0-11, rather than 1-12.

Using this syntax, each field can be represented as a range of values indicating all times between the two times. For example, `2-6` in the `dayOfWeek` field indicates Monday through Friday, inclusive. Each field can also be specified as a comma-separated list. For instance, a minute field of `0,15,30,45` means every quarter hour on the quarter hour. Lastly, each field can be defined as both a set of individual values and ranges of values. For example, an hour field of `9-17,0` indicates between the hours of 9 A.M. and 5 P.M., and on the hour of midnight.

Additional semantics are as follows:

- If multiple schedules are supplied (using a semi-colon (;) as the separator), the next scheduled time for the set is determined using the schedule that returns the soonest value. One use for this is for specifying schedules that change based on the day of the week (see the final example below).
- A value of 1 (one) for the `dayOfWeek` equates to Sunday.
- A value of * means every time for that field. For instance, a * in the Month field means every month. A * in the Hour field means every hour.
- A value of `l` or `last` (not case sensitive) indicates the greatest possible value for a field.
- If a day-of-month is specified that exceeds the normal maximum for a month, then the normal maximum for that month will be specified. For example, if it is February during a leap year and 31 was specified, then the scheduler will

schedule as if 29 was specified instead. This means that setting the month field to 31 always indicates the last day of the month.

- If milliseconds are specified, they are rounded down to the nearest 50th of a second. The values are 0, 19, 39, 59, ..., 979, and 999. Thus, 0-40 gets rounded to 0-39 and 50-999 gets rounded to 39-999.

Note: When a Calendar is not supplied as a method parameter to one of the static methods in this class, the calendar used is a `java.util.GregorianCalendar` with a default `java.util.TimeZone` and a default `java.util.Locale`.

Table 4-3 Example Time-to-Deliver Schedules

Example	Description
`0 0 0,30 * * * *`	Exact next nearest half-hour
`* * 0,30 4-5 * * *`	Anytime in the first minute of the half hours in the 4 A.M. and 5 A.M. hours
`* * * 9-16 * * *`	Between 9 A.M. and 5 P.M. (9:00.00 A.M. to 4:59.59 P.M.)
`* * * * 8-14 * 2`	The second Tuesday of the month
`* * * 13-16 * * 0`	Between 1 P.M. and 5 P.M. on Sunday
`* * * * * 31 *`	The last day of the month
`* * * * 15 4 1`	The next time April 15th occurs on a Sunday
`0 0 0 1 * * 2-6;0 0 0 2 * * 1,7`	1 A.M. on weekdays; 2 A.M. on weekends

JMS Schedule Interface

The `weblogic.jms.extensions.schedule` class has methods that will return the next scheduled time that matches the recurring time expression. This expression uses the same syntax as the `TimeToDeliverOverride`. The time returned in milliseconds can be relative or absolute.

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

You can define the next scheduled time after the *given* time using the following method:

```
public static Calendar nextScheduledTime(
    String schedule,
    Calendar calendar
  ) throws ParseException {
```

You can define the next scheduled time after the *current* time using the following method:

```
public static Calendar nextScheduledTime(
    String schedule,
      ) throws ParseException {
```

You can define the next scheduled time after the *given* time in absolute milliseconds using the following method:

```
public static long nextScheduledTimeInMillis(
    String schedule,
    long timeInMillis
  ) throws ParseException
```

You can define the next scheduled time after the *given* time in relative milliseconds using the following method:

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

You can define the next scheduled time after the *current* time in relative milliseconds using the following method:

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

Interaction with the Time-to-Live Value

If the specified time-to-live value (`JMSExpiration`) is less than or equal to the specified time-to-deliver value, then the message delivery succeeds. However, the message is then silently expired.

Managing Connections

The following sections describe how to manage connections:

Defining a Connection Exception Listener

An exception listener asynchronously notifies an application whenever a problem occurs with a connection. This mechanism is particularly useful for a connection waiting to consume messages that might not be notified otherwise.

Note: The purpose of an exception listener is not to monitor all exceptions thrown by a connection, but to deliver those exceptions that would not be otherwise be delivered.

You can define an exception listener for a connection using the following `Connection` method:

```
public void setExceptionListener(
  ExceptionListener listener
) throws JMSException
```

You must specify an `ExceptionListener` object for the connection.

The JMS Provider notifies an exception listener, if one has been defined, when it encounters a problem with a connection using the following `ExceptionListener` method:

```
public void onException(
  JMSException exception
)
```

The JMS Provider specifies the exception that describes the problem when calling the method.

You can access the exception listener for a connection using the following Connection method:

```
public ExceptionListener getExceptionListener(
) throws JMSException
```

Accessing Connection Metadata

You can access the metadata associated with a specific connection using the following `Connection` method:

```
public ConnectionMetaData getMetaData(
) throws JMSException
```

This method returns a `ConnectionMetaData` object that enables you to access JMS metadata. The following table lists the various type of JMS metadata and the get methods that you can use to access them.

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

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

Starting, Stopping, and Closing a Connection

To control the flow of messages, you can start and stop a connection temporarily using the `start()` and `stop()` methods, respectively, as follows.

The `start()` and `stop()` method details are as follows:

```
public void start(
) throws JMSException

public void stop(
) throws JMSException
```

A newly created connection is stopped—no messages are received until the connection is started. Typically, other JMS objects are set up to handle messages before the connection is started, as described in "Setting Up a JMS Application" on page 4-4. 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 5-5.
- Does not force an acknowledge of client-acknowledged sessions. By not forcing an acknowledge, no messages are lost for queues and durable subscriptions that require reliable processing.

When you close a connection, all associated objects are also closed. You can continue to use the message objects created or received via the connection, except the received message's `acknowledge()` method. Closing a closed connection has no effect.

Note: Attempting to acknowledge a received message from a closed connection's session throws an `IllegalStateException`.

Managing Sessions

The following sections describe how to manage sessions, including:

- Defining a Session Exception Listener
- Closing a Session

Defining a Session Exception Listener

An exception listener asynchronously notifies a client in the event a problem occurs with a session. This is particularly useful for a session waiting to consume messages that might not be notified otherwise.

Note: The purpose of an exception listener is not to monitor all exceptions thrown by a session, only to deliver those exceptions that would otherwise be undelivered.

You can define an exception listener for a session using the following `WLSession` method:

```
public void setExceptionListener(
  ExceptionListener listener
) throws JMSException
```

You must specify an `ExceptionListener` object for the session.

The JMS Provider notifies an exception listener, if one has been defined, when it encounters a problem with a session using the following `ExceptionListener` method:

```
public void onException(
  JMSException exception
)
```

The JMS Provider specifies the exception encountered that describes the problem when calling the method.

You can access the exception listener for a session using the following `WLSession` method:

```
public ExceptionListener getExceptionListener(
) throws JMSException
```

Note: Because there can only be one thread per session, an exception listener and message listener (used for asynchronous message delivery) cannot execute simultaneously. Consequently, if a message listener is executing at the time a problem occurs, execution of the exception listener is blocked until the message listener completes its execution. For more information about message listeners, see "Receiving Messages Asynchronously" on page 4-30.

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 5-5.
- Does not force an acknowledge of client acknowledged sessions, ensuring that no messages are lost for queues and durable subscriptions that require reliable processing.

When you close a session, all associated producers and consumers are also closed.

Note: If you want to issue the `close()` method within an `onMessage()` method call, the system administrator must select the Allow Close In OnMessage check box when configuring the connection factory. For more information, see "JMS Connection Factory Tasks" 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 available in `weblogic.jms.extensions`:

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

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

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

Each method updates the following:

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

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

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

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

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

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

You can then call the `findQueue()` method after the `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.

Deleting Destinations Dynamically

You can dynamically delete JMS destinations (queue or topic) using:

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

The JMS server removes the deleted destination in real time, therefore, it's not necessary to redeploy the JMS server for the deletion to take effect.

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

Preconditions for Deleting Destinations

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

- The destination must not be a member of a distributed destination. For more information, see "Using Distributed Destinations" on page 4-90.
- The destination must not be the error destination for some other destination. For more information, see "Configuring an Error Destination for Undelivered Messages" on page 4-36.

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

Using the JMSHelper Class Methods

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

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

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

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

Semantics When Deleting Destinations

When a destination is deleted using the Administration Console, the JMSHelper function, or a user-defined JMX application, the following behaviors and semantics apply.

Producer, Consumer, and Browser Creation

Once a destination is deleted, applications will no longer be able to create producers, consumers, or browsers for the deleted destination. Any attempt to do so will result in the application receiving an `InvalidDestinationException` — as if the destination does not exist.

CLosing of Consumers

All existing consumers for the deleted destination are closed. The closing of a consumer generates a `ConsumerClosedException`, which is delivered to the `ExceptionListener`, if any, of the parent session. The message for the exception will read “Destination was deleted”.

When a consumer is closed, if it has an outstanding `receive()` operation, then that operation is cancelled and the caller receives a `null` indicating that no message is available. Attempts by an application to do anything but `close()` a closed consumer will result in an `IllegalStateException`.

Closing of Browsers

All browsers for the deleted destination are closed. Attempts by an application to do anything but `close()` a closed browser will result in an `IllegalStateException`. Closing of a browser implicitly closes all enumerations associated with the browser.

Closing of Enumerations

All enumerations for the deleted destination are closed. The behavior after an enumeration is closed depends on the last call before the enumeration was closed. If a call to `hasMoreElements()` returns a value of *true*, and no subsequent call to `nextElement()` has been made, then the enumeration guarantees that next element can be enumerated. This produces the specifics.

When the last call before the close was to `hasMoreElements()`, and the value returned was *true*, then the following behaviors apply:

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

If a given enumeration has never been called, or the last call before the close was to `nextElement()`, or the last call before the close was to `hasMoreElements()` and the value returned was false, then the following behaviors apply:

- Calls to `hasMoreElements()` will return false.
- Calls to `nextElement()` will throw a `NoSuchElementException`.

Cancelled Blocking Send Operations

All blocking send operations posted against the deleted destination are cancelled. Send operations waiting for quota will receive a `ResourceAllocationException`.

For more information on using blocking send operations, see "Avoiding Quota Exceptions by Blocking Message Producers" in the *Administration Console Online Help*.

Affected Transactions

The deletion of a destination does not affect existing uncommitted transactions. Any uncommitted work associated with a deleted destination is allowed to complete as part of the transaction. However, since the destination is deleted, the net result of all operations (rollback, commit, etc.) is the deletion of the associated messages.

Physical Deletion of Existing Messages

Under normal operating circumstances all durable subscribers for the deleted destination are permanently deleted. All messages, persistent and non-persistent, stored in the deleted destination are permanently removed from the messaging system.

Timestamps for Troubleshooting Deleted Destinations

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

However, if a persistent message somehow has a version number that is *newer* than the version number in the `config.xml` for the recreated destination, then either the system clock was rolled back when the destination was deleted and recreated (while the JMS server was not running), or a different `config.xml` is being used. In this situation, the JMS server will fail to boot. To save the persistent message, you can set the version number (the `CreationTime` field) in the `config.xml` to match the version number in the persistent message. Otherwise, you can change the version number in the `config.xml` so that it is newer than the version number in the persistent message; this way, the JMS server can delete the message when it is rebooted.

Statistics

Statistics for the deleted destination and the hosting JMS server are updated as the messages are physically deleted. However, the deletion of some messages can be delayed pending the outcome of some other operation. This includes messages sent and/or received in a transaction, as well as unacknowledged non-transactional messages received by a client.

Using Temporary Destinations

Temporary destinations enable an application to create a destination, as required, without the system administration overhead associated with configuring and creating a server-defined destination.

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 4-57.

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

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

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

Creating a Temporary Queue

You can create a temporary queue using the following `QueueSession` method:

```
public TemporaryQueue createTemporaryQueue(
) throws JMSException
```

For example, to create a reference to a `TemporaryQueue` that will exist only for the duration of the current connection, use the following method call:

```
QueueSender = Session.createTemporaryQueue();
```

Creating a Temporary Topic

You can create a temporary topic using the following `TopicSession` method:

```
public TemporaryTopic createTemporaryTopic(
) throws JMSException
```

For example, to create a reference to a temporary topic that will exist only for the duration of the current connection, use the following method call:

```
TopicPublisher = Session.createTemporaryTopic();
```

Deleting a Temporary Destination

When you finish using a temporary destination, you can delete it (to release associated resources) using the following `TemporaryQueue` or `TemporaryTopic` method:

```
public void delete(
) throws JMSException
```

Setting Up Durable Subscriptions

WebLogic JMS supports durable and non-durable subscriptions.

For durable subscriptions, WebLogic JMS stores a message in a persistent file or database until the message has been delivered to the subscribers or has expired, even if those subscribers are not *active* at the time that the message is delivered. A subscriber is considered active if the Java object that represents it exists. Durable subscriptions are supported for Pub/Sub messaging only.

Note: Durable subscriptions cannot be created for distributed topics. However, you can still create a durable subscription on distributed topic member and the other topic members will forward the messages to the member that has the durable subscription. For more information on using distributed topics, see "Using Distributed Destinations" on page 4-90.

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
- Managing 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 Factory Tasks" 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 "Configuring JMS" in the *Administration Console Online Help*.

  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 4-70. 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 4-58.

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

Deleting Durable Subscriptions

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

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

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

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

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

Note: You can also delete durable subscriptions from the Administration Console. For information on managing durable subscriptions, see "Managing Durable Subscriptions" on page 4-62.

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 4-61.

   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 4-60 to recreate a durable subscription of the same name, but specifying a different topic name, message selector, or `noLocal` value.

   The durable subscription is recreated based on the new values.

Note: When recreating a durable subscription, be careful to avoid creating a durable subscription with a duplicate name. For example, if you attempt to delete a durable subscription from a JMS server that is unavailable, the delete call fails. If you subsequently create a durable subscription with the same name on a different JMS server, you may experience unexpected results when the first

JMS server becomes available. Because the original durable subscription has not been deleted, when the first JMS server again becomes available, there will be two durable subscriptions with duplicate names.

Managing Durable Subscriptions

You can monitor and delete durable subscriptions from the Administration Console. For more information, see "Configuring JMS" in the *Administration Console Online Help*.

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 4-70.

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 2-15.

The following table lists the Message class set and get methods for each of the supported data types.

Note: In some cases, the `send()` method overrides the header field value set using the `set()` method, as indicated in the following table.

Header Field	Set Method	Get Method

`JMSCorrelationID`	`public void setJMSCorrelationID( String correlationID ) throws JMSException`	`public String getJMSCorrelationID( ) throws JMSException`
		
		`public byte[] getJMSCorrelationIDAsBytes( ) throws JMSException`

`JMSDestination`[1]	`public void setJMSDestination( Destination destination ) throws JMSException`	`public Destination getJMSDestination( ) throws JMSException`
`JMSDeliveryMode`[1]	`public void setJMSDeliveryMode( int deliveryMode ) throws JMSException`	`public int getJMSDeliveryMode( ) throws JMSException`
`JMSDeliveryTime`[1]	`public void setJMSDeliveryTime( long deliveryTime ) throws JMSException`	`public long getJMSDeliveryTime( ) throws JMSException`

Header Field	Set Method	Get Method
`JMSDeliveryMode`[1]	`public void setJMSDeliveryMode( int deliveryMode ) throws JMSException`	`public int getJMSDeliveryMode( ) throws JMSException`

`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 pre-WebLogic JMS 6.0 and 6.1 `JMSMessageID` formats:
	`public void oldJMSMessageIDToNew( String id, long timeStamp ) throws JMSException`
	
	`public void newJMSMessageIDToOld( String id, long timeStamp ) throws JMSException`
		`public String getJMSMessageID( ) throws JMSException`

`JMSPriority`[1]	`public void setJMSPriority( int priority ) throws JMSException`	`public int getJMSPriority( ) throws JMSException`
`JMSRedelivered`[1]	`public void setJMSRedelivered( boolean redelivered ) throws JMSException`	`public boolean getJMSRedelivered( ) throws JMSException`
`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`

Header Field	Set Method	Get Method
JMSType	public void setJMSType(String type) throws JMSException	public String getJMSType( ) throws JMSException

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

The `examples.jms.sender.SenderServlet` example, provided with WebLogic Server in the *`WL_HOME`*`\samples\server\src\examples\jms\sender` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation, shows how to set header fields in messages that you send and how to display message header fields after they are sent.

For example, the following code, which appears after the `send()` method, displays the message ID that was assigned to the message by WebLogic JMS:

```
System.out.println("Sent message " +
       msg.getJMSMessageID() + " to " +
       msg.getJMSDestination());
```

Setting Message Property Fields

To set a property field, call the appropriate set method and specify the property name and value. To read a property field, call the appropriate get method and specify the property name.

The sending application can set properties in the message, and the receiving application can subsequently view them. The receiving application cannot change the properties without first clearing them using the following `clearProperties()` method:

```
public void clearProperties(
) throws JMSException
```

This method does not clear the message header fields or body.

Note: The `JMSX` property name prefix is reserved for JMS. The connection metadata contains a list of JMSX properties, which can be accessed as an enumerated list using the `getJMSXPropertyNames()` method. For more information, see "Accessing Connection Metadata" on page 4-45.

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

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

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

Data Type	Set Method	Get Method
short	`public void setShortProperty( String name, short value ) throws JMSException`	`public short getShortProperty( String name ) throws JMSException`
String	`public void setStringProperty( String name, String value ) throws JMSException`	`public String getStringProperty( String name ) throws JMSException`

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

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

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

This method returns all property field names as an enumeration. You can then retrieve the value of each property field by passing the property field name to the appropriate get method, as described in the previous table, based on the property field data type.

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

Table 4-5 Message Property Conversion Chart

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

Table 4-5 Message Property Conversion Chart (Continued)

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

You can test whether or not a property value has been set using the following `Message` method:

```
public boolean propertyExists(
  String name
) throws JMSException
```

You specify a property name and the method returns a boolean value indicating whether or not the property exists.

For example, the following code sets two String properties and an int property:

```
msg.setStringProperty("User", user);
msg.setStringProperty("Category", category);
msg.setIntProperty("Rating", rating);
```

For more information about message property fields, see "Message Property Fields" on page 2-19 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 4-70.

Once you have defined a queue, you can access the queue name and message selector associated with a queue browser using the following `QueueBrowser` methods:

```
public Queue getQueue(
) throws JMSException

public String getMessageSelector(
) throws JMSException
```

In addition, you can access an enumeration for browsing the messages using the following `QueueBrowser` method:

```
public Enumeration getEnumeration(
) throws JMSException
```

The `examples.jms.queue.QueueBrowser` example, provided with WebLogic Server in the *`WL_HOME`*`\samples\server\src\examples\jms\queue` directory, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation, shows how to access the header fields of received messages.

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

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

The following provides an excerpt from the `displayQueue()` method defined in the `QueueBrowser` example. In this example, the `QueueBrowser` object is used to obtain an enumeration that is subsequently used to scan the queue's messages.

```
   public void displayQueue(
   ) throws JMSException
  {
```

```
Enumeration e = qbrowser.getEnumeration();
Message m = null;

if (! e.hasMoreElements()) {
  System.out.println("There are no messages on this queue.");
} else {

  System.out.println("Queued JMS Messages: ");
  while (e.hasMoreElements()) {
    m = (Message) e.nextElement();
    System.out.println("Message ID " + m.getJMSMessageID() +
                  " delivered " + new Date(m.getJMSTimestamp())
                       " to " + m.getJMSDestination());
  }
}
```

When a queue browser is no longer being used, you should close it to free up resources. For more information, see "Releasing Object Resources" on page 4-33.

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 4-62 and "Setting Message Property Fields" on page 4-65, respectively.

Defining Message Selectors Using SQL Statements

A message selector is a boolean expression. It consists of a String with a syntax similar to the `where` clause of an `SQL select` statement.

The following excerpts provide examples of selector expressions.

```
salary > 64000 and dept in ('eng','qa')
```

```
(product like 'WebLogic%' or product like '%T3')
       and version > 3.0
```

```
hireyear between 1990 and 1992
       or fireyear is not null
```

```
fireyear - hireyear > 4
```

The following example shows how to set a selector when creating a queue receiver that filters out messages with a priority lower than 6.

```
String selector = "JMSPriority >= 6";
qsession.createReceiver(queue, selector);
```

The following example shows how to set the same selector when creating a topic subscriber.

```
String selector = "JMSPriority >= 6";
qsession.createSubscriber(topic, selector);
```

For more information about the message selector syntax, see the `javax.jms.Message` Javadoc.

Defining XML Message Selectors Using XML Selector Method

For XML message types, in addition to using the SQL selector expressions described in the previous section to define message selectors, you can use the following method:

```
String JMS_BEA_SELECT(String type, String expression)
```

`JMS_BEA_SELECT` is a built-in function in WebLogic JMS SQL syntax. You specify the syntax type, which must be set to `xpath` (XML Path Language) and an XPath expression. The XML path language is defined in the XML Path Language (XPath) document, which is available at the XML Path Language Web site at: `http://www.w3.org/TR/xpath`

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

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

```
String sel = "JMS_BEA_SELECT('xpath',
'/order/item[2]/name/text()') = 'Mitre Saw'";
```

Pay careful attention to the use of double and single quotes and spaces. Note the use of single quotes around `xpath`, the XML tab, and the string value.

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

```
String sel = "JMS_BEA_SELECT('xpath',
'/order/item[3]/id/text()') = '66'";
```

Displaying Message Selectors

You can use the following `MessageConsumer` method to display a message selector:

```
public String getMessageSelector(
) throws JMSException
```

This method returns either the currently defined message selector or null if a message selector is not defined.

Defining Server Session Pools

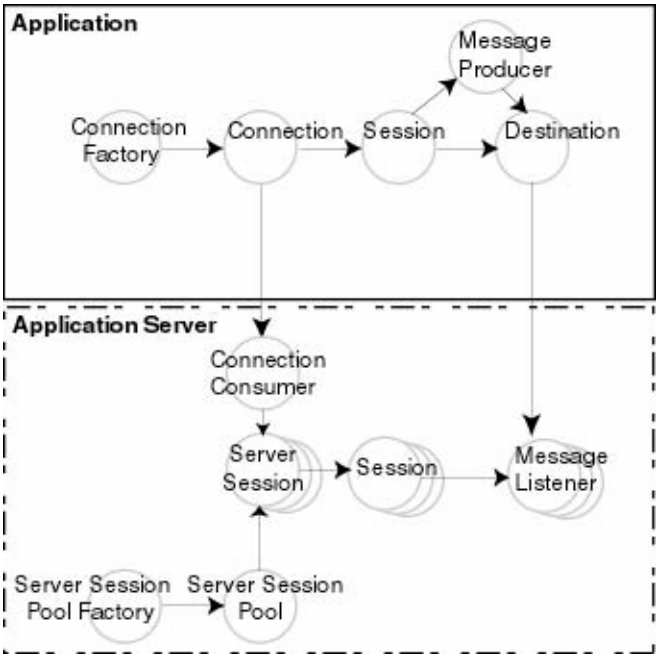
WebLogic JMS implements an optional JMS facility for defining a server-managed pool of server sessions. This facility enables an application to process messages concurrently.

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 4-3 Server Session Pool Facility

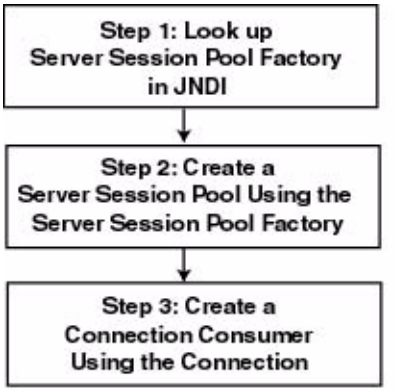


As illustrated in the figure, the application provides a single-threaded message listener. The connection consumer, implemented by JMS on the application server, performs the following tasks to process one or more messages:

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 4-4 Preparing for Concurrent Message Processing



Applications can use other application server providers' session pool implementations within this flow. Server session pools can also be implemented using message-driven beans. For information on using message driven beans to implement server session pools, see "Designing Message-Driven Beans" in *Programming WebLogic Enterprise JavaBeans*.

If the session pool and connection consumer were defined during configuration, you can skip this section. For more information on configuring server session pools and connection consumers, see Configuring JMS in the *Administration Console Online Help*.

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

Step 1: Look Up Server Session Pool Factory in JNDI

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

WebLogic JMS defines one `ServerSessionPoolFactory` object, by default: `weblogic.jms.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 2-21 or the `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 2-21 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 "Configuring JMS" in the *Administration Console Online Help*
- Including in your application the Connection methods described in the following sections

For more information about the `ConnectionConsumer` class, see "ConnectionConsumer" on page 2-22 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 4-70.

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 4-70.

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 4-57.

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

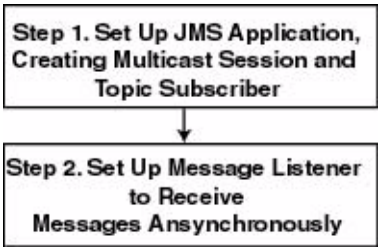
The limitations of multicasting include:

- Multicast messages are not guaranteed to be delivered to all members of the host group. For messages requiring reliable delivery and recovery, you should not use multicasting.
- For interoperability with different versions of WebLogic Server, clients cannot have an earlier release of WebLogic Server installled than the host. They must all have at least the same version or higher.

For an example of when multicasting might be useful, consider a stock ticker. When accessing stock quotes, timely delivery is more important than reliability. When accessing the stock information in real-time, if all or a portion of the contents is not delivered, the client can simply request the information to be resent. Clients would not want to have the information recovered, in this case, as by the time it is redelivered, it would be out-of-date.

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

Figure 4-5 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 4-87.

- For each destination, the Multicast Address (IP), Port, and TTL (Time-To-Live) attributes are specified. To better understand the TTL attribute setting, see "Example: Multicast TTL" on page 4-88.

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 "JMS Topic --> Configuration --> Multicast" in 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 non-durable subscribers. An attempt to create a durable subscriber on a multicast session will cause a `JMSException` to be thrown.

For example, the following method illustrates how to create a multicast session for the Pub/Sub messaging model.

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

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

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.

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.

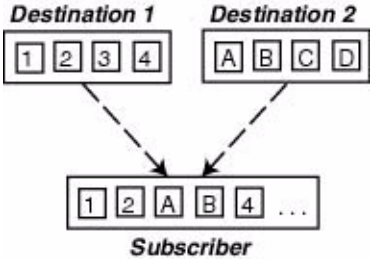
Set up the message listener for the topic subscriber, as described in "Receiving Messages Asynchronously" on page 4-30.

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 4-6 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 4-6 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 Destination Tasks" in the *Administration Console Online Help*.

Example: Multicast TTL

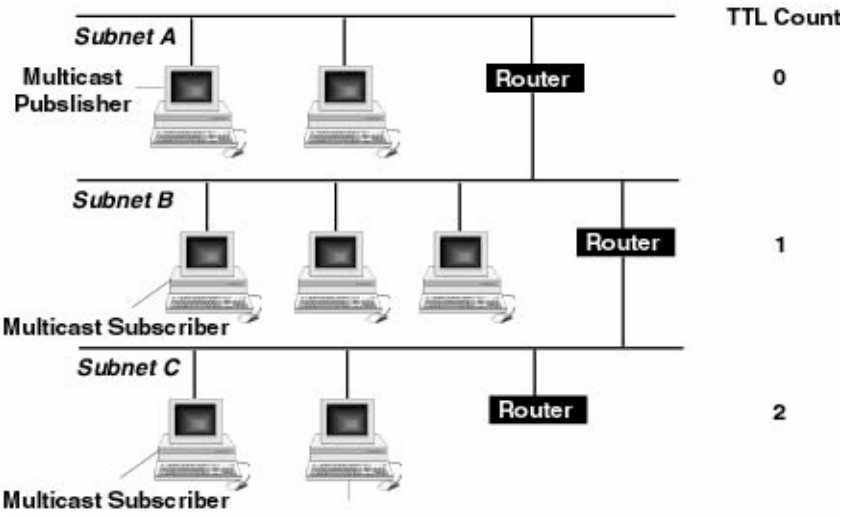
Note: The following example is a very simplified illustration of how the Multicast TTL (time-to-live) destination configuration attribute impacts the delivery of messages across routers. It is strongly advised that you seek the assistance of your network administrator when configuring the multicast TTL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

Consider the following network diagram.

Figure 4-7 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.

Using Distributed Destinations

By enabling you to configure multiple physical destinations (queues and topics) as members of a single distributed destination set, WebLogic JMS supports service continuity in the event of a WebLogic Server failure within a cluster. Once properly configured, your producers and consumers are able to send and receive messages through the distributed destination. WebLogic JMS then distributes the messaging load across all available destination members within the distributed destination. When a member becomes unavailable due a server failure, traffic is then redirected toward other available destination members in the set.

For instructions on configuring distributed destinations using the Administration Console, see "Distributed Destination Tasks" in the *Administration Console Online Help*.

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

- Accessing Distributed Destinations
- Accessing Distributed Destination Members
- Load Balancing Messages Across a Distributed Destination
- Distributed Destination Migration

Accessing Distributed Destinations

A distributed destination is actually a set of physical JMS destination members (queues or topics) that is accessed through a single JNDI name. As such, a distributed destination can be looked up using JNDI. It implements the `javax.jms.Destination` interface, and can be used to create producers, consumers, and browsers.

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

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

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

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

Looking Up Distributed Queues

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

The queue members can be located anywhere, but must all be served by JMS servers in a single server cluster. When a message is sent to a distributed queue, it is sent to exactly one of the physical queues in the set of members for the distributed queue. Once the message arrives at the queue member, it is available for receipt by consumers of that queue member only.

Note: Queue members can forward messages to other queue members by configuring the Forward Delay attribute in the Administration Console, which is disabled by default. This attribute defines the amount of time, in seconds, that a distributed queue member with messages, but which has no consumers, will wait before forwarding its messages to other queue members that do have consumers.

QueueSenders

After creating a queue sender, if the queue supplied at creation time was a distributed queue, then each time a message is produced using the sender a decision is made as to which queue member will receive the message. Each message is sent to a single physical queue member.

The message is not replicated in any way. As such, the message is only available from the queue member where it was sent. If that physical queue becomes unavailable before a given message is received, then the message is unavailable until that queue member comes back online.

It is not enough to send a message to a distributed queue and expect the message to be received by a queue receiver of that distributed queue. Since the message is sent to only one physical queue member, there must be a queue receiver receiving or listening on that queue member.

Note: For information on the load-balancing heuristics for distributed queues with zero consumers, see "Load Balancing Heuristics" on page 4-98.

QueueReceivers

When creating a queue receiver, if the supplied queue is a distributed queue, then a single physical queue member is chosen for the receiver at creation time. The created `QueueReceiver` is pinned to that queue member until the queue receiver loses its access to the queue member. At that point, the consumer will receive a `JMSException`, as follows:

- If the queue receiver is synchronous, then the exception is returned to the user directly.
- If the queue receiver is asynchronous, then the exception is delivered inside of a `ConsumerClosedException` that is delivered to the `ExceptionListener` defined for the consumer session, if any.

Upon receiving such an exception, an application can close its queue receiver and recreate it. If any other queue members are available within the distributed queue, then the creation will succeed and the new queue receiver will be pinned to one of those queue members. If no other queue member is available, then the application won't be able to recreate the queue receiver and will have to try again later.

Note: For information on the load-balancing heuristics for distributed queues with zero consumers, see "Load Balancing Heuristics" on page 4-98.

QueueBrowsers

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

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

Looking Up Distributed Topics

A distributed topic is a set of physical JMS topic members. As such, a distributed topic can be used to create a `TopicPublisher` and `TopicSubscriber`. The fact that a distributed topic represents multiple physical topics is mostly transparent to the application.

Note: Durable subscribers (`DurableTopicSubscriber`) cannot be created for distributed topics. However, you can still create a durable subscription on distributed topic member and the other topic members will forward the messages to the topic member that has the durable subscription.

The topic members can be located anywhere but must all be served either by a single WebLogic Server or any number of servers in a cluster. When a message is sent to a distributed topic, it is sent to all of the topic members in the distributed topic set. This allows all subscribers to the distributed topic to receive messages published for the distributed topic.

A message published directly to a topic member of a distributed destination (that is, the publisher did not specify the distributed destination) is also forwarded to all the members of that distributed topic. This includes subscribers that originally subscribed to the distributed topic, and which happened to be assigned to that particular topic member. In other words, publishing a message to a specific distributed topic member automatically forwards it to all the other distributed topic members, just as publishing a message to a distributed topic automatically forwards it to all of its distributed topic members. For more information about looking up specific distributed destination members, see "Accessing Distributed Destination Members" on page 4-95.

Deploying Message-Drive Beans on a Distributed Topic

When an MDB is deployed on a distributed topic and is targeted to a WebLogic Server instance in a cluster that is hosting two members of the distributed topic on a JMS server, the MDB gets deployed on both the members of the distributed topic. This occurs because MDBs are pinned to a distributed topic member's destination name.

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

TopicPublishers

When creating a topic publisher, if the supplied destination is a distributed destination, then any messages sent to that distributed destination are sent to all available topic members for that distributed topic, as follows:

- If one or more of the distributed topic members is not reachable, and the message being sent is non-persistent, then the message is sent only to the available topic members.
- If one or more of the distributed topic members is not reachable, and the message being sent is persistent, then the message is stored and forwarded to the other topic members when they become reachable. However, the message can only be persistently stored if the topic member has a JMS store configured.

  Note: Every effort is made to first forward the message to distributed members that utilize a persistent store. However, if none of the distributed members utilize a store, then the message is still sent to one of the members according to the selected load-balancing algorithm, as described in "Load Balancing Messages Across a Distributed Destination" on page 4-96.

- If all of the distributed topic members are unreachable (regardless of whether the message is persistent or non-persistent), then the publisher receives a `JMSException` when it tries to send a message.

TopicSubscribers

When creating a topic subscriber, if the supplied topic is a distributed topic, then the topic subscriber receives messages published to that distributed topic.

If one or more of the topic members for the distributed topic are not reachable by a topic subscriber, then depending on whether the messages are persistent or non-persistent the following occurs:

- Any persistent messages published to one or more unreachable distributed topic members are eventually received by topic subscribers of those topic members

once they become reachable. However, the messages can only be persistently stored if the topic member has a JMS store configured.

- Any non-persistent messages published to those unreachable distributed topic members will not be received by that topic subscriber.

Ultimately, a topic subscriber is pinned to a physical topic member. If that topic member becomes unavailable, then the topic subscriber will receive a `JMSException`, as follows:

- If the topic subscriber is synchronous, then the exception is returned to the user directly.
- If the topic subscriber is asynchronous, then the exception is delivered inside of a `ConsumerClosedException` that is delivered to the `ExceptionListener` defined for the consumer session, if any.

Upon receiving such an exception, an application can close its topic subscriber and recreate it. If any other topic member is available within the distributed topic, then the creation should be successful and the new topic subscriber will be pinned to one of those topic members. If no other topic member is available, then the application will not be able to recreate the topic subscriber and will have to try again later.

Accessing Distributed Destination Members

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

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

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

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

Load Balancing Messages Across a Distributed Destination

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

For more information about configuring load balancing for a distributed destination, see "Configuring Message Load Balancing" in the *Administration Console Online Help*.

Load Balancing Options

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

- Round-Robin Distribution
- Random Distribution

Round-Robin Distribution

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

If weights are assigned to any of the physical destinations in the set for a given distributed destination, then those physical destinations appear multiple times in the ordering. For instance, if the weights of destinations A, B and C are 2, 5, and 3 respectively, then the ordering will be A, B, C, A, B, C, B, C, B, B. That is, a number

of passes are made through the basic ordering (A, B, C). The number of passes is equal to the highest weight of the destinations within the set. On each pass, only those destinations with a weight that is greater than or equal to the ordinal value of the pass are included in the ordering. Following this logic, this example would produce the following results:

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

Random Distribution

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

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

Consumer Load Balancing

When an application creates a consumer, it must provide a destination. If that destination represents a distributed destination, then WebLogic JMS must find a physical destination that consumer will receive messages from. The choice of which destination member to use is made by using one of the load-balancing algorithms described in "Load Balancing Options" on page 4-96. The choice is made only once: when the consumer is created. From that point on, the consumer gets messages from that member only.

Producer Load Balancing

When a producer sends a message, WebLogic JMS looks at the destination where the message is being sent. If the destination is a distributed destination, WebLogic JMS makes a decision as to where the message will be sent. That is, the producer will send to one of the destination members according to one of the load-balancing algorithms described in "Load Balancing Options" on page 4-96.

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

Note: If a producer attempts to send a persistent message to a distributed destination, every effort is made to first forward the message to distributed members that utilize a persistent store. However, if none of the distributed members utilize a persistent store, then the message will still be sent to one of the members according to the selected load-balancing algorithm.

Load Balancing Heuristics

In addition to the algorithms described in "Load Balancing Options" on page 4-96, WebLogic JMS uses the following heuristics when choosing an instance of a destination.

- Transaction Affinity
- Server Affinity
- Queues with Zero Consumers

Transaction Affinity

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

Server Affinity

When a WebLogic Server attempts to load balance consumers or producers across physical destinations in a distributed destination set, it will first attempt to load balance across any physical destinations that are also running on the same WebLogic Server.

Note: For more information about configuring server affinity for a distributed destination, see "Configuring Server Affinity" in the *Administration Console Online Help*.

Queues with Zero Consumers

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

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

Defeating Load Balancing

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

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

JNDI Lookup

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

CreateQueue() and CreateTopic()

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

Connection Factories

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

For more information about configuring load balancing for a distributed destination, see "Enabling Message Load Balancing" in the *Administration Console Online Help*.

Distributed Destination Migration

For JMS implementations that take use the WebLogic Server 7.0 service migration feature, when a JMS server fails it can migrate to another WebLogic Server within a cluster—along with all of its distributed destination members. However, the target WebLogic Server may already be hosting a JMS server with all of its physical destinations. This can lead to situations where the same WebLogic Server hosts two physical destinations for a single distributed destination. This is permissible in the short term, since a WebLogic Server can host multiple physical destinations for that distributed destination. However, load balancing in this situation is less effective.

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

For more information about the configuring JMS migratable targets, see "Configuring JMS Migratable Targets" on page 3-10.

Distributed Destination Failover

When a JMS server migrates to another WebLogic Server due a server failure, consumers that are pinned to a failed destination member must be closed and recreated.

For more information about procedures for recovering from a WebLogic Server failure, see "Recovering from a WebLogic Server Failure" on page 3-15.

CHAPTER

5 Using Transactions with WebLogic JMS

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

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

Overview of Transactions

A transaction enables an application to coordinate a group of messages for production and consumption, treating messages sent or received as an atomic unit.

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

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

For example, when shopping online, you select items and store them in an online shopping cart. Each ordered item is stored as part of the transaction, but your credit card is not charged until you confirm the order by checking out. At any time, you can cancel your order and empty your cart, rolling back all orders within the current transaction.

There are three ways to use transactions with JMS:

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

To enable multiple JMS servers in the same JTA user transaction, or to combine JMS operations with non-JMS operations (such as EJB), the two-phase commit license is required. For more information, see "Using JTA User Transactions" on page 5-5.

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 4-47.

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

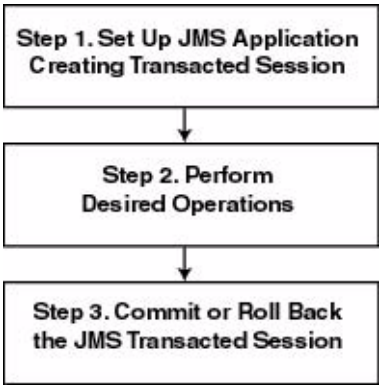
A JMS transacted session supports transactions that are located within the session. A JMS transacted session's transaction will not have any effects outside of the session. For example, rolling back a session will roll back all sends and receives on that session, but will not roll back any database updates.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

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 roll back the transaction.

To commit the transaction, execute the following method:

```
public void commit(
) throws JMSException
```

The `commit()` method commits all messages sent or received during the current transaction. Sent messages are made visible, while received messages are removed from the messaging system.

To roll back the transaction, execute the following method:

```
public void rollback(
) throws JMSException
```

The `rollback()` method cancels any messages sent during the current transaction and returns any messages received to the messaging system.

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

Using JTA User Transactions

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

You program your JTA user transaction applications using the `javax.transaction.UserTransaction` object to begin, commit, and roll back the transactions. When mixing JMS and EJB within a JTA user transaction, you can also start the transaction from the EJB, as described in "Transactions in EJB Applications" in *Programming WebLogic JTA*.

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

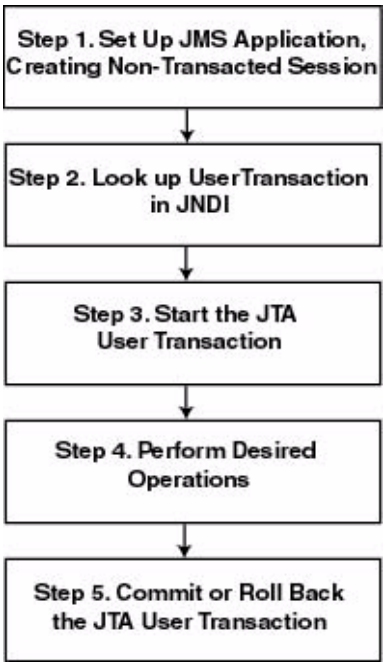
WebLogic Server supports the two-phase commit protocol (2PC), enabling an application to coordinate a single JTA transaction across two or more resource managers. It guarantees data integrity by ensuring that transactional updates are committed in all of the participating resource managers, or are fully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

Note: A separate 2PC transaction license is required to support this protocol. For transaction migration considerations related to 2PC, see "Porting WebLogic JMS Applications" on page 8-1.

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 check box, as described in "JMS Connection Factories" in the *Administration Console Online Help*.

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

```
UserTransaction xact =
ctx.lookup("javax.transaction.UserTransaction");
```

Step 3: Start the JTA User Transaction

Start the JTA user transaction using the `UserTransaction.begin()` method. For example:

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

Step 4: Perform Desired Operations

Perform the desired operations associated with the current transaction.

Step 5: Commit or Roll Back the JTA User Transaction

Once you have performed the desired operations, execute one of the following methods to commit or roll back 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 roll back 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 "Designing Message-Driven Beans" in *Programming WebLogic EJB*.

Example: JMS and EJB in a JTA User Transaction

The following example shows how to set up an application for mixed EJB and JMS operations in a JTA user transaction by looking up a `javax.transaction.UserTransaction` using JNDI, and beginning and then committing a JTA user transaction. In order for this example to run, the User Transactions Enabled check box must be selected when the system administrator configures the connection factory.

Note: In addition to this simple JTA User Transaction example, refer to the example provided with WebLogic JTA, located in the *`WL_HOME`*`\samples\server\src\examples\jta\jmsjdcb` directory, where where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation.

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

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

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

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

Step 1 Set up the JMS application, creating a non-transacted session. For more information on setting up the JMS application, refer to "Setting Up a JMS Application" on page 4-4.

```
//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 Using WebLogic JMS with EJBs and Servlets

The following sections describe features in WebLogic Server 8.1 that make it easier to use WebLogic JMS in conjunction with J2EE components, such as a servlet or an EJB (Enterprise Java Bean).

Overview

This release of WebLogic Server makes it easier to use WebLogic JMS in conjunction with servlets or EJBs. These usability features are generally hidden behind the J2EE standard, but they have been enhanced for this release. Using this support should be considered as the "best practice" way to send a WebLogic JMS message from inside an EJB or servlet.

The "Foreign JMS Provider Support" on page 6-10 section briefly describes the new console support for foreign JMS providers, as documented in Accessing Foreign JMS Providers section of the *Administration Console Online Help*. This feature makes it

possible to map foreign JMS providers — including instances of WebLogic Server in another cluster or domain — so that they appear in the local JNDI tree as a local JMS object.

J2EE Support for WebLogic JMS

The WebLogic Server 8.1 makes it easier to use WebLogic JMS inside a J2EE component by providing usability features, such as:

- Automatic pooling of JMS connection and session objects (and some pooling of message producer objects as well).
- Automatic transaction enlistment for JMS providers that support XA.
- Testing of the JMS connection and reestablishment after a failure.
- Security credentials that are managed by the container.

These features are accessed from inside an EJB or a servlet by declaring a WebLogic JMS connection factory as a *`resource`* in the deployment descriptors. An example of this is provided in "Referencing a JMS Connection Factory" on page 6-3. Once a connection factory is registered as a `resource`, then the application can look it up from JNDI using the `java:comp/env/` subtree that is created for each EJB or servlet. It is important to note that these features are only enabled when using a resource inside the deployment descriptors. Writers of EJBs and servlets still have direct access to the JMS provider by performing a direct JNDI lookup of the connection factory.

For more information about packaging EJBs, see "Packaging EJBs for the WebLogic Server Container" in *Programming WebLogic Enterprise JavaBeans*. For more information about programming servlets, see "Programming Tasks" in *Programming WebLogic HTTP Servlets*.

Referencing a JMS Connection Factory

A JMS connection factory can be registered as part of an EJB or servlet by including a `resource-ref` element in the `ejb-jar.xml` or `web.xml` file. In other words, WebLogic Server 8.1 creates a "wrapped" JMS connection factory that provides the other, more advanced features described in this section.

Here is an example of such an element:

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

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

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

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

In addition to this element, there must be a matching `resource-description` element in the `weblogic-ejb-jar.xml` or `weblogic.xml` file that tells the J2EE container which JMS connection factory to put in that location. Here is an example:

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

The connection factory specified here must already exist in the global JNDI tree. This example uses one of JMS connection factories that are automatically created whenever the built-in WebLogic JMS server is used. To use another WebLogic JMS connection factory from the same cluster, simply include that connection factory's JNDI name inside the `jndi-name` element. To use a connection factory from another vendor, or from another WebLogic Server cluster, create a Foreign JMS Server, as described in "Accessing Foreign JMS Providers" in the *Administration Console Online Help*.

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

Referencing a JMS Destination

It is also possible to bind a JMS destination (a queue or topic) into the `java:comp/env` JNDI tree. This feature is useful for consistency, and to make an application less dependent on a particular configuration of WebLogic Server. To do this, there must be a `resource-env-ref` element in the `weblogic-ejb-jar.xml` or `web.xml` file, as follows:

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

There must also be a matching `resource-env-description` element in the `weblogic-ejb-jar.xml` or `weblogic.xml` file, as follows:

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

Again, if the destination does not exist, then the application is deployed, but there will be an exception thrown when trying to use the destination.

Sending a Message

Once the resources have been mapped to the `java:comp/env` JNDI tree, then they can be used inside an EJB or a servlet. For instance, the following code fragment sends a message:

```
InitialContext ic = new InitialContext();
QueueConnectionFactory qcf =
  (QueueConnectionFactory)ic.lookup("java:comp/env/jms/QCF");
Queue destQueue =
  (Queue)ic.lookup("java:comp/env/jms/TESTQUEUE");
ic.close();
QueueConnection connection = qcf.createQueueConnection();
try {
  QueueSession session = connection.createQueueSession(0, false);
  QueueSender sender = session.createSender(destQueue);
  TextMessage msg = session.createTextMessage("This is a test");
  sender.send(msg);
} finally {
```

```
  connection.close();
}
```

This is standard code that complies with the J2EE specification and which should run on any EJB product that properly supports J2EE — the difference is that it runs more efficiently on WebLogic Server 8.1, because under the covers various objects are pooled, as described in "Pooled Session Objects" on page 6-8.

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

Also, none of the transactional XA extensions to the JMS API are being used in this code fragment. Instead, the container uses them internally if the JMS code is used inside a transaction context. But whether XA is being used or not internally, the user-written code is the same, and does not use any of the JMS XA classes. This is what is specified by J2EE. By writing EJB code in this way, EJBs can run either in an environment where transactions are present or in a non-transactional environment, just by changing the deployment descriptors.

Under the Covers

This section explains what is happening "under the covers" when WebLogic Server creates a set of wrappers around the JMS objects. For example, in the code fragment provided in "Sending a Message" on page 6-4, since the JMS connection factory was looked up from the `java:comp/env` JNDI tree, the actual JMS connection factory is not being returned, but an instance of a WebLogic-specific wrapper class. This wrapper object intercepts certain calls to the JMS provider and inserts the correct J2EE behavior, as described in the following sections.

Automatically Enlisting Transactions

If a wrapped JMS connection is used to send or receive a message inside a transaction context, then the JMS session being used to send or receive the message is automatically enlisted in the transaction using the XA capabilities of the JMS provider.

This is the case whether the transaction was started implicitly because the JMS code was invoked inside an EJB with container-managed transactions enabled, or if the transaction was started manually using the `UserTransaction` interface in a servlet or an EJB that supports bean-managed transactions.

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

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

In order to send or receive a message inside a transaction using a JMS provider that does not support XA, either declare the EJB with a transaction mode of `NotSupported`, or suspend the transaction using one of the JTA APIs.

For more information on the attributes available when configuring a WebLogic JMS connection factory that supports transactions, see "JMS Connection Factory --> Configuration --> Transactions" in the *Administration Console Online Help*.

Container-Managed Security

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

Container — When the `res-auth` element is set to `Container`, then security to the JMS provider is managed by the J2EE container. In this case, if the JMS connection factory was mapped into the JNDI tree using a Foreign JMS Connection Factory configuration MBean, then the user name and password from that MBean is used (see "Foreign JMS Provider Support" on page 6-10). Otherwise, WebLogic Server connects to the provider with no user name or password specified. In this mode, it is an error to pass a user name and password to the `createConnection()` method of the JMS connection factory.

Application — When the `res-auth` element is set to `Application`, then any user name or password on the MBean is ignored. Instead, the application code must specify a user name and password to the `createConnection()` method of the JMS connection factory, or use the version of this function with no user name or password if none are required.

Connection Testing

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

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

J2EE Compliance

The J2EE specification states that you should not be allowed to make certain JMS API calls inside a J2EE application. The JMS wrappers enforce these restrictions by throwing an exception when they are violated. They are as follows:

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

Furthermore, the `createSession()` method, and the associated `createQueueSession()` and `createTopicSession()` methods, are handled differently. This method takes two parameters: an "acknowledgement" mode and a "transacted" flag. When used inside an EJB, these two parameters are ignored. If a transaction is present, then the JMS session is enlisted in the transaction as described in "Automatically Enlisting Transactions" on page 6-5; otherwise, it is not. By default, the acknowledgement mode is set to "auto acknowledge". This behavior is expected by the J2EE specification. (This may make it more difficult to receive messages from inside an EJB, but the recommended way to receive messages from inside an EJB is to use a message-driven bean.)

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

Pooled Session Objects

The JMS wrappers pool various session objects in order to make code like the example provided in "Sending a Message" on page 6-4 more efficient. A pooled JMS connection is a session pool used by EJBs and servlets that use a *resource-reference* element in their EJB deployment descriptor to define their JMS connection factories.

Pooled JMS sessions can be monitored using the Server --> Monitoring --> JMS node on the Administration Console. For more information, see "Server --> Monitoring --> JMS" in the *Administration Console Online Help*.

Improving Performance

The automatic pooling of connections and other objects by the JMS wrappers means that it is efficient to write code as shown in "Sending a Message" on page 6-4. Although in this example the Connection Factory, Connection, and Session objects are created every time a message is sent, in reality these three classes work together so that when used as shown, they do little more than retrieve a Session object from the pool.

Speeding Up JNDI Lookups

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

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

Inside an EJB, these lookups can be performed inside the `ejbCreate()` method and assigned to an instance variable. For a session bean, each instance of the bean will then have its own copy, but this is perfectly fine. Since stateless session beans are pooled, this is also very efficient, and is perfectly consistent with the J2EE specifications. (Whereas, caching these objects in a static member of the EJB class may work, but it is discouraged by the J2EE specification.)

However, if these objects are cached inside the `ejbCreate()` or `init()` method, then the EJB or servlet must have some way to recreate them if there has been a failure. This is necessary because some JMS providers, like WebLogic JMS, may invalidate a Destination object after a server failure. So, if the EJB runs on *Server A*, and JMS runs on *Server B*, then the EJB on *Server A* will have to perform the JNDI lookup of the objects from *Server B* again after that server has recovered. The example, "PoolTestCMPBean.java" on page 6-13 includes a sample EJB that performs this caching and relookup process correctly.

Speeding Up Object Creation

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

However, this technique will improve performance for critical code, since the management of the JMS session pool does add overhead. If you want to use this technique, you must make sure that you close and reopen the JMS connection and session objects after a server failure; otherwise, your EJB or servlet will not be able to access the JMS provider after it has been restarted.

Using the Right Transaction Mode

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

However, performing the transaction enlistment has overhead. Furthermore, if an XA connection factory is used, but the `send()` or `receive()` method is invoked outside a transaction, the container must still create a JTA transaction to wrap the `send()` or

`receive()` method in order to ensure that the operation properly takes place no matter which JMS provider is used. Although this is only a one-phase commit, it can still slow down the server.

Therefore, when writing an EJB or servlet that uses a JMS resource in a non-transactional manner, then it is best to use a JMS connection factory that is not configured to support XA. For more information on configuring a WebLogic JMS connection factory, see "Configuring a JMS Connection Factory" in the *Administration Console Online Help*.

Foreign JMS Provider Support

Another set of features for WebLogic Server 8.1 makes it possible to create a "symbolic link" between a JMS connection factory or destination object in an external JNDI provider to an object inside the local WebLogic Server. There are three configuration MBeans for this task:

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

For instructions on configuring these MBeans with the Administration Console, refer to "Accessing Foreign JMS Providers" in the *Administration Console Online Help*.

Once deployed, these MBeans work by creating objects in the server's JNDI tree, which perform the lookup of the remote object every time they are looked up. This means that the local server and the remote JNDI directory are never out of sync. However, it means that a JNDI lookup of one of these MBeans can potentially be expensive. The sections on "Pooled Session Objects" on page 6-8 describes some ways around this.

Examples of JMS Wrapper Functions

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

ejb-jar.xml

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

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

<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>PoolTestCMPBean</ejb-name>
   <home>weblogic.jms.pool.test.PoolTestCMPHome</home>
   <remote>weblogic.jms.pool.test.PoolTestCMP</remote>
   <ejb-class>weblogic.jms.pool.test.PoolTestCMPBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>

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

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


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

</ejb-jar>
```

weblogic-ejb-jar.xml

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

<!DOCTYPE weblogic-ejb-jar PUBLIC
  "-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN"
  "http://www.bea.com/servers/wls700/dtd/weblogic-ejb-jar.dtd">

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>PoolTestCMPBean</ejb-name>
    <stateless-session-descriptor>
      <pool>
        <max-beans-in-free-pool>8</max-beans-in-free-pool>
        <initial-beans-in-free-pool>2</initial-beans-in-free-pool>
      </pool>
    </stateless-session-descriptor>

    <reference-descriptor>
      <resource-description>
        <res-ref-name>jms/QCF</res-ref-name>
        <jndi-name>weblogic.jms.XAConnectionFactory</jndi-name>
      </resource-description>
      <resource-env-description>
        <res-env-ref-name>jms/TESTQUEUE</res-env-ref-name>
```

```
        <jndi-name>TESTQUEUE</jndi-name>
      </resource-env-description>
    </reference-descriptor>
    <jndi-name>PoolTestCMP</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

PoolTestCMP.java

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

import java.rmi.*;
import javax.ejb.*;
public interface PoolTestCMP extends EJBObject
{
  public String sendXATransactional(String queue,
                                    String text, int count)
    throws RemoteException;
}
```

PoolTestCMPHome.java

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

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

public interface PoolTestCMPHome
  extends EJBHome
{
  PoolTestCMP create()
    throws CreateException, RemoteException;
}
```

PoolTestCMPBean.java

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

import java.lang.reflect.*;
import java.rmi.*;
```

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

import weblogic.deployment.jms.*;

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

  public void ejbActivate()
  {
  }

  public void ejbRemove()
  {
  }

  public void ejbPassivate()
  {
  }

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

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

  public void ejbCreate()
    throws CreateException
```

```
  {
    try {
      lookupJNDIObjects();
    } catch (NamingException ne) {
      throw new CreateException(ne.toString());
    }
  }

  public String sendXATransactional(String queue,
String text, int count)
    throws RemoteException
  {
    String id = "Not sent yet";
    try {
      if ((qcf == null) || (destination == null)) {
        lookupJNDIObjects();
      }
      QueueConnection connection = qcf.createQueueConnection();
      try {
        QueueSession session = connection.createQueueSession
                                                    (false, 0);
        TextMessage message = session.createTextMessage
                                                    ("Testing");
        QueueSender sender = session.createSender(destination);
        sender.send(message);
        id = message.getJMSMessageID();
      } finally {
        connection.close();
      }
    } catch (Exception e) {
      // Invalidate the JNDI objects if there is a failure
      // this is necessary because the destination object
      // may become invalid if the destination server has
      // been shut down
      qcf = null;
      destination = null;
      throw new RemoteException("Failure in EJB: " + e);
    }
    return id;
  }
}
```

CHAPTER

7 WebLogic JMS Thin Client

The following sections describe how to deploy and use the WebLogic JMS thin client:

Overview

While the size of the `weblogic.jar` file may not be a problem when running server-side applications, it does cause a very large footprint for today's enterprise client-server applications that may be running thousands of clients. Having to deploy the full 20+ MB `weblogic.jar` file along with a client application can significantly increase the size of the deployed application, possibly making it too big to be practical (such as the case with a Java applet-based client program).

At around 400 KB, the thin client `wljmsclient.jar` file provides a smaller client footprint by using a client-side library that contains only the set of supporting files required by client-side programs. The JMS thin client also requires using the standard `wlclient.jar` (around 300 KB), which has the base client support for clustering, security, and transactions.

The thin client `.jar` files are supported only with the JRE 1.4.x or later. No classes other than the JRE and any user-defined classes are required on the client machine. The thin client is based upon the RMI-IIOP protocol stack available in JRE 1.4.x. The basics of making RMI requests are handled by the JRE, enabling a significantly smaller client. Client-side development is performed using standard J2EE APIs, rather than WebLogic Server APIs.

The thin client `.jar` files are located in the *`WL_HOME`*`\server\lib` subdirectory of the WebLogic Server installation directory (for example, `c:\bea\weblogic81b\server\lib`).

For more information on using WebLogic Servers client applications, see "Understanding WebLogic Server Applications" in *Developing WebLogic Server Applications*.

Benefits of Using the JMS Thin Client

Although small in size, the WebLogic JMS and standard WebLogic Server thin clients provides the following functionality to client applications and applets:

- Full WebLogic JMS functionality is available—both standard JMS and WebLogic Server extensions—except for client-side XML selection for multicast sessions and the JMSHelper class methods.
- EJB (Enterprise Java Bean) access.
- JNDI access.
- RMI access (indirectly used by JMS).
- SSL access (using JSSE in JRE 1.4.x).
- Transaction capability.
- Clustering capability.
- HTTP/HTTPS tunneling.
- Fully internationalized.

Limitations of Using the JMS Thin Client

The WebLogic JMS thin client does not provide all of the functionality of the normal `weblogic.jar`, as follows:

- JDBC
- JMX

Deploying the JMS Thin Client

Deployment of the WebLogic JMS thin client depends on the following requirements:

- Each client must have the JRE 1.4.x installed.
- The thin client `.jar` files are located in the *`WL_HOME`*`\server\lib` subdirectory of the WebLogic Server installation directory (for example, `c:\bea\weblogic81b\server\lib`).
- The thin JMS `wljmsclient.jar` and standard `wlclient.jar` files must be in each client's classpath and installed somewhere on the client's file system.
- The WebLogic JMS thin client requires using the RMI over IIOP standard for communicating between client and server.
  - URLs using `t3` or `t3s` will transparently use `iiop` or `iiops`
  - URLs using `http` or `https` will transparantly use `iiop` tunneling.
- Adhere to J2EE programming guidelines, in particular the use of `PortableRemoteObject.narrow()` rather than using casts.

CHAPTER

8 Porting WebLogic JMS Applications

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

Existing Feature Functionality Changes

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

- Existing Feature 5.1 to 6.0 Functionality Changes
- Existing Feature 6.0 to 6.1 Functionality Changes

Existing Feature 5.1 to 6.0 Functionality Changes

The following table lists the changes in existing feature functionality from WebLogic Server version 5.1, and also indicates any code changes that might be required as a result. For additional information pertaining to the JMS Specification's version change history, refer to Chapter 11, "Change History" in the specification.

Category	Description	Code Modification

Connection Factories	Two default connection factories have been deprecated. The JNDI names for these factories are:
	■ `javax.jms.QueueConnectionFactory`
	■ `javax.jms.TopicConnectionFactory`
	For backwards compatibility, the JNDI names for these two connection factories are still defined and supported.
	
	WebLogic JMS 6.*x* or later defines one connection factory, by default:
	`weblogic.jms.ConnectionFactory`
	You can also specify user-defined connection factories using the *Administration Console*.
	Note: Using the default connection 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 check boxes when configuring the connection factory.
		For more information about defining connection factories, see "JMS Connection Factory Tasks" 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 check box when configuring the WebLogic Server.	None required. This is a configuration requirement. For more information, see "Server --> Services --> JMS" in the *Administration Console* Online Help.

Category	Description	Code Modification
Connections	When closing a connection, the call blocks until outstanding synchronous calls and asynchronous listeners have completed.	None required.
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 check box is selected when configuring the connection factory. If the acknowledge mode is `AUTO_ACKNOWLEDGE`, the current message will still be automatically acknowledged.	None required. This is a configuration requirement. For more information, see "JMS Connection Factory Tasks" in the *Administration Console Online Help*.

Category	Description	Code Modification

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

Category	Description	Code Modification
Destinations	The `createQueue()` and `createTopic()` methods do not create destinations dynamically, only references to destinations that already exist given the vendor-specific destination name.	Update any portion of code that uses `createQueue()` or `createTopic()` to dynamically create destinations using the following JMSHelper class methods, respectively: `createPermanentQueueAsync()` and `createPermanentTopicAsync()`. For example, if your code used the following method to dynamically create a queue: `queue=qsession.createQueue(queueName);` You should modify the code to dynamically create a queue, as described in the sample `findQueue()` method in "Using the JMSHelper Class Methods" on page 4-50. For more information, see "Creating Destinations Dynamically" on page 4-49.
	When creating temporary destinations, you must specify a temporary template.	None required. This is a configuration requirement. For more information, see "JMS Template Tasks" 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 4-58.

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.*x* and later Javadoc, as they are system interfaces and should not be used within client applications.	If used, remove any references to these objects from the client application.
Transactions	To combine JMS and EJB database calls within the same transaction, a two-phase commit (2PC) license is required. In previous releases of WebLogic Server, it was possible to combine them by using the same database connection pool.	None required.
	Recovering or rolling back received queue messages makes them available to all consumers on the queue. In previous releases of WebLogic Server, rolled back messages were only available to the session that rolled back the message, until that session was closed.	None required.

Existing Feature 6.0 to 6.1 Functionality Changes

The following table lists the changes in existing feature functionality from WebLogic Server 6.0, and also indicates any code changes that might be required as a result. For additional information pertaining to the JMS Specification's change history, see Chapter 11, "Change History," of Sun Microsystem's JMS Specification

Category	Description	Code Modification
Connection Factories	For the Acknowledge Policy attribute in the Administration Console, the new default value of `All` is a work-around to accommodate a change in the JMS Specification. This new default setting represents a change from prior versions of JMS, which internally defaulted to `Previous`, and which did not appear as an option in the Administration Console. As the message acknowledge policy for the connection factory, the Acknowledge Policy attribute only applies to implementations that use the `CLIENT_ACKNOWLEDGE` mode for a non-transacted session. ■ `All` — acknowledge all messages ever received by a given session, regardless of which message calls the acknowledge method. ■ `Previous` — acknowledge all messages received by a given session, but only up to and including the message that calls the acknowledge method. For more information on message acknowledge modes, refer to "Non-transacted Session" on page 2-9. **Note:** For connection factories used by MDBs (message-driven beans), always set the Acknowledge Policy field to `Previous`. Although the default MDB connection factory already does this, foreign connection factories may not.	If you want to acknowledge only previously received messages, up to and including the message that calls the acknowledge method, change the default Acknowledge Policy setting from `All` to `Previous` via the JMS --> Connection Factory --> General tab in the Administration Console.

Category	Description	Code Modification
Destinations	In WLS 6.0, the JMS documentation correctly specifies values of `default`, `true`, and `false` for the `StoreEnabled` attribute of the `JMSDestinationMBean`, even though the software allowed for mixed case characters. version 6.1 or later, however, requires all lowercase characters for the `StoreEnabled` settings.	None required. This is a configuration requirement. For more information, see "JMS Template Tasks" in the *Administration Console* Online Help.

Porting Existing Applications

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

- Porting Steps for 4.5 and 5.1 Applications to 6.x
- Porting Steps for 6.0 Applications to 6.1
- Porting Steps for 6.x Applications to 7.0

Before You Begin

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

- Weblogic Server 4.5.1 — Porting is supported *only* for SP15. Customers running all service packs should contact BEA Support.
- Weblogic Server 5.1 — Customers running SP07 or SP08 should contact BEA Support before porting existing JDBC stores to version 7.0.

- In order to port object messages, the object classes need to be in the Weblogic Server 7.0 server `CLASSPATH`.
- For destinations that are not configured in Weblogic Server 7.0, the ported messages will be dropped and the event will be logged.

- WebLogic Server 6.*x* — All applications are supported in version 7.0. However, if you want your applications to take advantage of the new highly available JMS features, you will need to configure your existing physical destinations (queues and topics) to be part of a single distributed destination set. For more information, see "Using Distributed Destinations" in *Programming WebLogic JMS*.

Porting Steps for 4.5 and 5.1 Applications to 6.x

Before you can use an existing WebLogic JMS 6.*x* application, you must port the WebLogic Server versions 4.5 and 5.1 configuration and message data as follows:

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

   Warning: Abruptly stopping the old version of WebLogic Server while messaging is still in process may cause problems during porting. Processing should be inactive before shutting down the old server and beginning the porting to WebLogic Server 6.*x*.

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

3. Ported configuration information using the configuration conversion facility.

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

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

   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 5.1 to 6.0 Functionality Changes" on page 8-2.

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

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

   a. Make a backup of the existing JDBC database.

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

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

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

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

      Both the existing and new database information will exist on disk while the porting is performed, doubling the space requirements. Once porting is complete, you can delete the old JDBC database stores, as described in "Deleting JDBC Database Stores" on page 8-12.

5. Update any existing code, as required, to reflect the feature functionality changes described in "Existing Feature 5.1 to 6.0 Functionality Changes" on page 8-2.

6. Start up the WebLogic Server and the existing JDBC database stores will be ported automatically.

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

Porting Steps for 6.0 Applications to 6.1

Before you can use an existing WebLogic JMS 6.*x* application, you must port the WebLogic Server 6.0 configuration and message data as follows

1. Check the connection factory configuration for version 6.0. You may need to modify programs that call the version 6.1 default connection factory so that they load one of the following connection factories:
   - One of the version 6.0 default connection factories.
   - A custom connection factory.
2. Properly shut down the version 6.0 WebLogic Server before beginning the porting process.

   Warning: Abruptly stopping the old version of WebLogic Server while messaging is still in process may cause problems during porting. Processing should be inactive before shutting down the old server and beginning the porting to WebLogic Server 6.*x*.
3. Upgrade the WebLogic Server environment, as described in *Installing WebLogic Server*.
4. Update any existing code, as required, to reflect the feature functionality changes described in "Existing Feature 5.1 to 6.0 Functionality Changes" on page 8-2.

   Warning: Before starting the version 6.1 WebLogic Server, you may want to backup your version 6.0 stores. This is because version 6.0 servers *cannot* use 6.1 stores, and any attempts to do so may cause data corruption.
5. Start the version 6.1 WebLogic Server. This server will continue to use the previous version 6.0 stores.

Porting Steps for 6.x Applications to 7.0

All WebLogic JMS 6.*x* applications are supported in version 7.0. However, if you want your applications to take advantage of the new highly available JMS features, you will need to configure your existing physical destinations (queues and topics) to be part of a single distributed destination set.

For more information on using JMS distributed destinations, see "Using Distributed Destinations" in *Programming WebLogic JMS*.

Deleting JDBC Database Stores

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

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

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

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

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

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, see "Configuring JMS" in 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 check box under the Configuration—Transactions tab on the Connection Factories node

JMS Transactions

To support JMS transacted sessions, configure the following:

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

Message Delivery

To define message delivery attributes, configure the following:

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

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

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 4-57.

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 4-73.

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 4-57.

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

Overview

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

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

About JMS Tables

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

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

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

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

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

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

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

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

Regenerating JDBC Database Stores

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

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

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

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

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

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

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

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

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

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

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

1. Set the WLS samples environment:

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

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

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

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

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

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

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

Index

D

E

F

H

J

M

R

S

T

U

X