BEA WebLogic Server®

Configuring and Managing WebLogic JMS

Version 9.0
Revised: July 22, 2005

Contents

Introduction and Roadmap

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

Document Scope and Audience

This document is a resource for system administrators responsible for configuring, managing, and monitoring WebLogic JMS resources, such as JMS servers, standalone destinations (queues and topics), distributed destinations, and connection factories.

The topics in this document are relevant to production phase administration, monitoring, or performance tuning topics. This document does not address the pre-production development or testing phases of a software project. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-2.

It is assumed that the reader is familiar with WebLogic Server system administration. This document emphasizes the value-added features provided by WebLogic Server JMS and key information about how to use WebLogic Server features and facilities to maintain WebLogic JMS in a production environment.

Guide to This Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.
- Chapter 2, "Understanding JMS Resource Configuration," provides an overview of WebLogic JMS architecture and features.
- Chapter 3, "Configuring JMS System Resources," describes how to configure basic WebLogic JMS resources, such as a JMS server, destinations (queues and topics), and connection factories.
- Chapter 4, "Configuring Clustered WebLogic JMS Resources," explains how to configure clustering JMS features, such as JMS servers, migratable targets, and distributed destinations.
- Chapter 5, "Configuring JMS Application Modules for Deployment," describes how to prepare JMS resources for an application module that can be deployed as a standalone resource that is globally available, or as part of an Enterprise Application that is available only to the enclosing application.
- Chapter 7, "Monitoring JMS Statistics and Managing Messages," describes how to monitor and manage the run-time statistics for your JMS objects from the Administration Console.
- Chapter 6, "Using WLST to Manage JMS Servers and JMS System Resources," explains how to use the WebLogic Scripting Tool to programmatically create and manage JMS resources.
- Chapter 9, "Tuning WebLogic JMS," explains how to get the most out of your applications by using the administrative performance tuning features available with WebLogic JMS.
- Chapter 8, "Troubleshooting WebLogic JMS," explains how to configure and manage message logs, and how to temporarily pause message operations on destinations.

Related Documentation

This document contains JMS-specific configuration and maintenance information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Programming WebLogic JMS* is a guide to JMS API programming with WebLogic Server.

- *Understanding WebLogic Server Clustering* explains how WebLogic Server clustering works
- *Deploying Applications to WebLogic Server* is the primary source of information about deploying WebLogic Server applications, which includes standalone or application-scoped JMS resource modules.
- "*Using the WebLogic Persistent Store*" in *Configuring WebLogic Server Environments* provides information about the benefits and usage of the system-wide WebLogic Persistent Store.
- *Configuring and Managing WebLogic Store-and-Forward* contains information about the benefits and usage of the Store-and-Forward service with JMS messages.
- *Configuring and Managing the WebLogic Messaging Bridge* explains how to configure a messaging bridge between any two messaging products—thereby, providing interoperability between separate implementations of WebLogic JMS, including different releases, or between WebLogic JMS and another messaging product.

JMS Samples and Tutorials for the JMS Administrator

In addition to this document, BEA Systems provides a variety of JMS code samples and tutorials that show JMS configuration and API use, and provide practical instructions on how to perform key JMS development tasks. BEA recommends that you run some or all of the JMS examples before configuring your own system.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample J2EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

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

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

JMS Examples in the WebLogic Server Distribution

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

Additional JMS Examples Available for Download

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

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

New and Changed JMS Features In This Release

WebLogic Server 9.0 introduces major changes in the configuration, deployment, and dynamic administration of WebLogic JMS.

- "JMS 1.1 Specification Support" on page 1-5
- Configuration and Administration Enhancements:
  - "Modular Configuration and Deployment of JMS Resources" on page 1-5
  - "Store-and-Forward for Highly Available Message Production" on page 1-6
  - A default, system-wide WebLogic Persistent Store
  - "Pause and Resume Message Operations on Destinations" on page 1-6
  - "Message Bridge Enhancements" on page 1-9
  - "Flexible and Simplified Destination Quotas" on page 1-8
- Message Management, Monitoring, and Diagnostics:
  - "Enhanced Run-time Message Management" on page 1-6

- – "More Transparency with Message Life Cycle Logging" on page 1-7
- – "Debug and Diagnostic Information More Readily Available" on page 1-7

- Functionality Enhancements
  - – "Strict Message Ordering with Unit-of-Order" on page 1-7
  - – "Access to JMS Applications from "C" Clients" on page 1-8
  - – "Message-Driven Bean (MDB) Enhancements for JMS" on page 1-8
  - – "Message ID Propagation Security Enhancement" on page 1-9
- Performance Improvements
  - – "Document Object Model (DOM) Support for XML Messages" on page 1-8
  - – "Automatic Compression of Messages" on page 1-10
  - – "Improved Message Paging" on page 1-9
- "Deprecated JMS Features, Methods, Interfaces, and Methods" on page 1-10

JMS 1.1 Specification Support

WebLogic Server 9.0 is compliant with the JMS 1.1 Specification for use in production, and so supports unified APIs that work for both queues and topics. For more information, see the Java JMS technology page on the Sun Web site at `http://java.sun.com/products/jms/`.

Modular Configuration and Deployment of JMS Resources

JMS configurations in WebLogic Server 9.0 are stored as modules, defined by an XML file that conforms to the `weblogic-jmsmd.xsd schema`, similar to standard J2EE modules. An administrator can create and manage JMS modules as global system resources, as modules packaged with a J2EE application (as a packaged resource), or as standalone modules that can be made globally available. With modular deployment of JMS resources, you can migrate your application and the required JMS configuration from environment to environment, such as from a testing environment to a production environment, without opening an enterprise application file (such as an EAR file) or a JMS standalone module, and without extensive manual JMS reconfiguration.

For more information, see "Configuring JMS System Resources" on page 3-1.

Store-and-Forward for Highly Available Message Production

The JMS Store-and-Forward feature is built on the WebLogic Store-and-Forward (SAF) service to provide highly available JMS message production. For example, a JMS message producer connected to a local server instance can reliably forward messages to a remote JMS destination, even though that remote destination may be temporarily unavailable when the message was sent. Persistent JMS messages are always forwarded with Exactly-once quality of service provided by the SAF. For non-persistent JMS messages, applications can also use the At-least-once and At-most-once qualities of service. JMS store-and-forward works with all WebLogic JMS features, including new features introduced in release 9.0.

For more information, see Understanding the Store-and-Forward Service in *Configuring and Managing WebLogic Store-and-Forward*.

Enhanced Run-time Message Management

New message administration enhancements greatly enhance a JMS administrator's ability to view and browse *all* messages, and to manipulate *most* messages in a running JMS Server, using either the Administration Console or through new public runtime APIs. These message management enhancements include message browsing (for sorting), message manipulation (such as move and delete), message import and export, as well as transaction management, durable subscriber management and JMS client connection management.

For more information on managing messages, see Chapter 7, "Monitoring JMS Statistics and Managing Messages."

Pause and Resume Message Operations on Destinations

New WebLogic JMS configuration and runtime APIs enable an administrator to pause and resume message production, message insertion (in-flight messages), and message consumption operations on a given JMS destination, or on all the destinations hosted by a single JMS Server, either programmatically or administratively. A potential use of this feature is asserting administrative control of the JMS subsystem behavior in the event of an external resource failure. Otherwise, such a failure could cause the JMS subsystem to ignore the external failures and overload the system (both server and clients) by continuously accepting and delivering (redelivering) messages.

For more information, see "Controlling Message Operations on Destinations" on page 8-15.

More Transparency with Message Life Cycle Logging

The message life cycle is an external view of the basic events that a JMS message will traverse through once it has been accepted by the JMS server, either through the JMS APIs or the JMS Message Management APIs. Therefore, Message Life Cycle Logging provides an administrator with better transparency about the existence of JMS messages from the JMS server viewpoint, in particular basic life cycle events, such as message production, consumption, and removal.

For more information, see "Message Life Cycle Logging" on page 8-7.

Debug and Diagnostic Information More Readily Available

In Weblogic Server 9.0, the JMS subsystem uses the new system-wide diagnostics service for centralized debug access and logging. With this new service, enabling debugging is easy, and debug and diagnostic information is more readily available so you can easily diagnose and fix problems.

For more information, see *Understanding the WebLogic Diagnostic Framework*.

Strict Message Ordering with Unit-of-Order

The Unit-of-Order feature goes beyond the message delivery ordering requirements in the JMS 1.1 Specification by providing JMS message producers with the ability to group ordered messages into a single unit. This single unit is called a *Unit-of-Order* and it *guarantees* that all messages created from that unit are processed sequentially in order by the same consumer until that consumer acknowledges them or it is closed. For example, if a queue has messages with many consumers, and each message has an account number as a Unit-of-Order, then two consumers will not process messages with the same account number at the same time. A Unit-of-Order can be created programmatically with new JMS API extensions to the JMSMessageProducer interface, or administratively by specifying a Unit-of-Order on a connection factory.

For more information, see Using Message Unit-of-Order in *Programming WebLogic JMS*.

Uniform Configuration of Distributed Destinations

This release of WebLogic Server introduces a new type of distributed destination, termed *Uniform Distributed Destination*, that greatly simplifies the management and development of distributed destination applications. By using the Uniform Distributed Destination feature, the administrator no longer needs to create or designate destination members, but relies on the system to uniformly create the necessary members on the JMS servers to which a JMS module is targeted. This feature ensures the consistent configuration of all distributed destination parameters, particularly in regards to weighting, security, persistence, paging, and quotas. The Weighted Distributed Destination feature is still available for those who want to manually fine-tune distributed destination members.

For more information, see "Configuring Distributed Destinations" on page 4-14.

Access to JMS Applications from "C" Clients

The Weblogic JMS C API enables programs written in "C" to participate in JMS applications. This implementation of the JMS C API uses JNI in order to access a Java Virtual Machine (JVM). The JMS C API does not support WebLogic JMS extensions, JMS Object messages, and Java MBean management.

For more information, see WebLogic C API in *Programming WebLogic JMS*.

Message-Driven Bean (MDB) Enhancements for JMS

MDB enhancements enable support for transaction batching (via processing multiple messages in a single transaction), and for load balancing MDB instances across member destinations in different clusters or domains, regardless of whether the MDB and destination reside in the same cluster or in different clusters or domains.

For more information, see Message-Driven Bean Enhancements in the *Release Notes*.

Document Object Model (DOM) Support for XML Messages

This feature enhances the WebLogic JMS API to provide native support for the Document Object Model (DOM) when sending XML messages. This will greatly improve performance for implementations that already use a DOM, since those applications will not have to flatten the DOM before sending XML messages.

For more information, see Sending XML Messages in *Programming WebLogic JM*S.

Flexible and Simplified Destination Quotas

The new destination quota objects for JMS modules allows administrators to configure named quota objects and then have destinations refer to them. To increase flexibility, destinations can be assigned their own quotas; multiple destinations can share a quota; or destinations can share the JMS server's quota. In addition, a destination that defines its own quota now no longer also shares space in the JMS server's quota. JMS servers still allow the direct configuration of message and byte quotas. However, these attributes are only used to provide quota for destinations that do not refer to a quota object.

For more information, see "Defining Quota" on page 9-1.

Improved Message Paging

The *message paging* feature for freeing up virtual memory during peak message load situations is always enabled on JMS servers. Additionally, administrators no longer need to create a dedicated message paging store since paged out messages can be stored in a directory on your file system. However, for the best performance you should specify that messages be paged to a directory other than the one used by the JMS server's persistent store.

For more information, see "Paging Out Messages To Free Up Memory" on page 9-6.

Message ID Propagation Security Enhancement

This enhancement introduces the JMSXUserID property, which provides the ability to identify the user who produced a message when the message is received. With this property, the recipient has the option of querying the message property to find out who sent the message, and then could possibly run different logic in the application, depending on who the initiator was. The message sender requests the user identity by using a connection factory with the AttachJMSXUserID attribute enabled. The delivery of the user identity is controlled by the Attach Sender value on destinations, which can also be managed for multiple destinations by using a JMS template.

Message Bridge Enhancements

The message bridge functionality has been improved with the following enhancements:

- Forwarded message IDs are always preserved for WebLogic JMS-to-WebLogic JMS communications.
- The introduction of the PreserveMsgProperty configuration parameter, which preserves the following properties for 9.0 target bridge destinations: message ID, priority, expiration time, message timestamp, user ID, delivery mode, priority, expiration time, redelivery limit, and unit-of-order name. However, only the delivery mode, priority, and expiration time will be preserved for pre-release 9.0 target bridge destinations.
- For easier message bridge adapter updates, WebLogic Server supports an exploded format of the RAR adapters (jms-xa-adp, jms-notran-adp, and jms-notran-adp51).

- The Administration Console features a Message Bridge Assistant for simplified configuration.

For more information, see "Configuring and Managing a Messaging Bridge" in *Configuring and Managing WebLogic Messaging Bridge*.

Automatic Compression of Messages

This feature enables the compression of messages that exceed a specified threshold size to improve the performance of sending messages travelling across JVM boundaries. A threshold can be set programmatically using a new JMS API extension to the JMSMessageProducer interface, or administratively by either specifying a value on a connection factory or on a SAF remote context. Once configured, message compression is triggered on producers for message sends, on connection factories for message receives and message browsing, or through SAF forwarding.

For more information, see "Compressing Messages" on page 9-5.

Deprecated JMS Features, Methods, Interfaces, and Methods

In WebLogic Server 9.0, many changes were made to the JMS subsystem, including the removal of some classes and the deprecation of many configuration MBeans.

Legacy JMS Resource Configuration Interfaces

The new module-based method of configuring WebLogic JMS resources uses Java Descriptor Bean interfaces to create JMS system modules and deployable packaged modules. This fundamental change necessitated the deprecation of the following MBean interfaces:

- JMSDestCommonMBean
- JMSTopicMBean
- JMSQueueMBean
- JMSConnectionFactoryMBean
- JMSTemplateMBean
- JMSDestinationKeyMBean
- ForeignJMSServerMBean
- ForeignJMSDestinationMBean

- ForeignJMSConnectionFactoryMBean
- JMSDistributedTopicMBean
- JMSDistributedTopicMemberMBean
- JMSDistributedTopicMBean
- JMSDistributedQueueMemberMBean

The new Descriptor Bean interfaces are documented in "WebLogic Server System Module Beans" in the *WebLogic Server MBean Reference*. The root bean that represents an entire JMS module is named JMSBean.

For more information about JMS module resources, see "What Are JMS Configuration Resources?" on page 2-5.

JMS Helper APIs

The module-based method of configuring JMS module resources necessitated the deprecation of the JMSHelper class for locating JMS runtime and configuration JMX MBeans.

This class has been replaced by a new JMSModuleHelper class with methods for managing (locate/create/delete) JMS Module configuration resources (descriptor beans) in a given JMS module (including the JMS Interop Module), and for locating JMS runtime MBeans.

For more information on using the JMS module helper methods, see the JMSModuleHelper Javadoc.

Messages Paging Enabled, Bytes Paging Enabled, and Paging Store Methods

The new simplified JMS Paging configuration necessitated the deprecation of the BytesPagingEnabled and MessagesPagingEnabled attributes of the JMSTemplateMBean, JMSDestinationMBean, and JMSServerMBean interfaces.

The new paging configuration has also caused the replacement of the PagingStore attribute on the JMSServerMBean with new PagingDirectory and MessageBufferSize attributes.

For more information, see "Paging Out Messages To Free Up Memory" on page 9-6.

Expiration Logging Policy Method

The new Message Life Cycle Logging feature necessitated the deprecation of the ExpirationLoggingPolicy attribute of the JMSTemplateMBean, JMSDestinationMBean, and JMSServerMBean interfaces.

For more information about expired message handling, see "Handling Expired Messages" on page 9-11. For more infomation about message life cycle logging, see "Message Life Cycle Logging" on page 8-7.

JMS Session Pool and JMS Connection Consumer Interfaces

The JMSSessionPoolMBean (and its associated methods on the JMSServerMBean) and JMSConnectionConsumerMBean interfaces have been deprecated. These interfaces were used to automatically create a JMS session pool and start the JMS consumers on the server side.

The ConnectionConsumer and ServerSessionPool APIs are still supported, but BEA strongly recommends using message-driven beans (MDBs), which are simpler, easier to manage, and more capable.

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

JMS Store Interfaces

The new WebLogic Persistent Store necessitated the deprecation of the following MBean interfaces:

- JMSStoreMBean
- JMSFileStoreMBean
- JMSJDBCStoreMBean

This deprecation also includes any associated JMS Store methods on the JMSServerMBean interface.

For more information, see Using The WebLogic Persistent Store in *Configuring WebLogic Server Environments*.

Pause and Resume Methods On the JMS Destination Runtime Interface

The new "Pause and Resume JMS Destinations" feature necessitated the deprecation of the `pause()`, `resume()`, and `isPaused()` APIs on the JMSDestinationRuntimeMBean interface. A new set of Pause/Resume APIs have been introduced to the JMSDestinationRuntimeMBean and JMSServerRuntimeMBean interfaces.

Message Purge Method on the JMS Durable Subscriber Runtime Interface

The new "JMS Message Management" feature necessitated the deprecation of the `purge()`, API on the JMSDurbableSubscriberRuntimeMBean interface.

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

JMS Extensions: WLMessage Interface

The following methods in the WLMessage class have been deprecated:

- `WLMessage.get/setDeliveryTime`
- `WLMessage.get/setRedeliveryLimit`

These methods have been replaced by the following properties:

- `javax.jms.Message.getIntProperty("JMS_BEA_DeliveryTime")`
- `javax.jms.Message.getIntProperty("JMS_BEA_RedeliveryLimit")`

These new properties includes the corresponding setter methods.

For more information on using these WLMessage extensions, see the JMS WLMessage Extension Javadoc.

Messaging Bridge Properties

The `Thread Pool Size` property for a server instance has been deprecated. Use the common thread pool or a work manager. See Change the Thread Pool Size in *Configuring and Managing the WebLogic Messaging Bridge*

CHAPTER 2

Understanding JMS Resource Configuration

These sections briefly review the different WebLogic JMS concepts and features, and describe how they work with other application components and WebLogic Server.

It is assumed the reader is familiar with Java programming and JMS 1.1 concepts and features.

Overview of JMS and WebLogic Server

The WebLogic Server implementation of JMS is an enterprise-class messaging system that is tightly integrated into the WebLogic Server platform. It fully supports the JMS 1.1 Specification and also provides numerous WebLogic JMS Extensions that go beyond the standard JMS APIs.

What Is the Java Message Service?

An enterprise messaging system enables applications to asynchronously 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 that is implemented by industry messaging providers. Specifically, JMS:

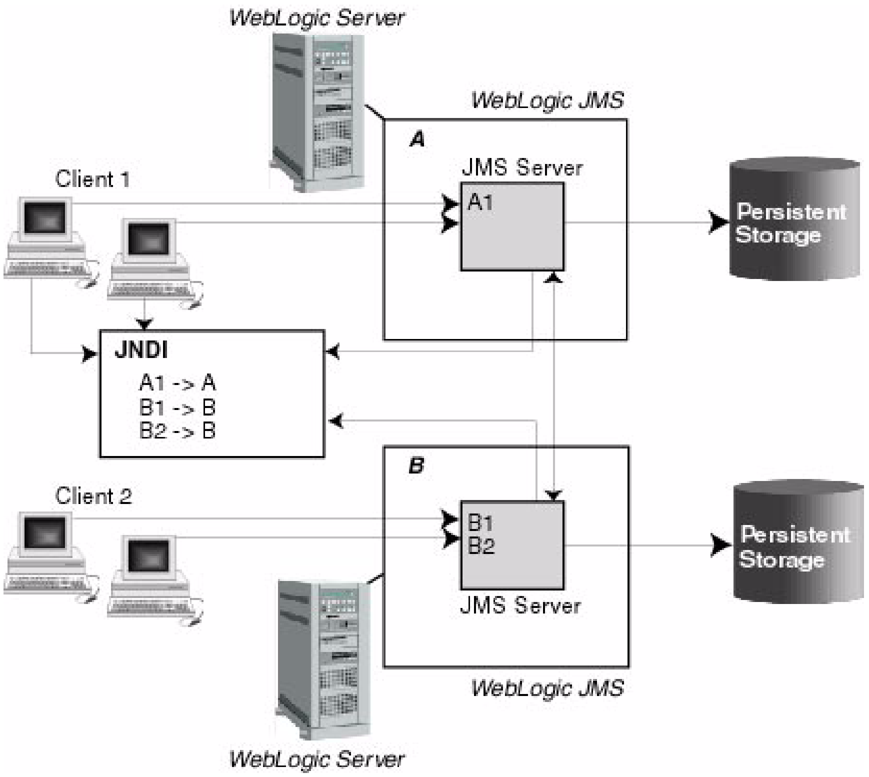
- Enables Java applications sharing a messaging system to exchange messages
- Simplifies application development by providing a standard interface for creating, sending, and receiving messages

WebLogic JMS accepts messages from *producer* applications and delivers them to *consumer* applications. For more information on JMS API programming with WebLogic Server, see *Programming WebLogic JMS*.

WebLogic JMS Anatomy and Environment

The following figure illustrates the WebLogic JMS architecture.

Figure 2-1 WebLogic JMS Architecture



where: A1 and B1 are connecton factories and B2 is a queue.

The major components of the WebLogic JMS architecture include:

- JMS servers host a defined set of destinations (queues for point-to-point communication or topics for publish-and-subscribe) in a JMS module, with which client applications can interact, and any associated persistent storage that resides on a WebLogic Server instance.
- JMS modules contain configuration resources (queue and topic destinations, connection factories, templates, destination keys, quota, distributed destinations, foreign servers, JMS store-and-forward), and are defined by XML documents that conform to the `weblogic-jmsmd.xsd` schema.

- Client JMS applications that either produce messages to destinations or consume messages from destinations.
- JNDI (Java Naming and Directory Interface), which provides a server *lookup* facility.
- WebLogic persistent storage (a server instance's default store, a user-defined file store, or a user-defined JDBC-accessible store) for storing persistent message data.

Domain Configuration: System Resources vs. Application Resources

In general, the WebLogic Server domain configuration file (`config.xml`) contains the configuration information required for a domain. This configuration information can be further classified into environment-related (or *system resource* definitions) and application-related information. Some examples of environment-related definitions are the identification and definition of JMS servers, JDBC data sources, WebLogic persistent stores, and server network addresses. These system resources are usually unique from domain to domain.

The configuration and management of these system resources are the responsibility of a WebLogic administrator, who usually receives this information from an organization's system administrator or MIS department. To accomplish these administrative tasks, an administrator can use the WebLogic Administration Console, various command-line tools, such as WebLogic Scripting Tool (WLST), or JMX APIs for programmatic administration.

Some examples of application-related definitions that are highly independent of the domain environment are the various J2EE application components configurations, such as EAR, WAR, JAR, RAR files, and which in this release includes JMS and JDBC modules. The application components are originally developed and packaged by an application development team, and may contain optional programs (compiled Java code) and respective configuration information (also called descriptors, which are mostly stored as XML files). In the case of JMS and JDBC modules, however, there are no compiled Java programs involved. These pre-packaged applications are given to WebLogic Server administrators for deployment in a WebLogic domain.

WebLogic Server provides tools for deploying applications, such as the Administration Console and the `weblogic.Deployer` command-line utility. The process of deploying an application links the application components to the environment-specific resource definitions, such as which server instances should host a given application component (targeting), and the WebLogic persistent store to use for persisting JMS messages.

Once the initial deployment is completed, an administrator has only limited control over deployed applications. For example, administrators are only allowed to ensure the proper life-cycle of these applications (deploy, undeploy, redeploy, remove, etc.) and to tune the parameters, such as increasing or decreasing the number of instances of any given application to satisfy the client needs. Other than lifecycle and tuning, any modification to these applications must be completed by the application development team.

What Are JMS Configuration Resources?

In prior releases, all JMS configuration information was stored in a WebLogic domain's configuration file. In this release, JMS configurations (destinations, connections factories, templates, etc.) are stored outside of the WebLogic domain as module descriptor files, which are defined by XML documents that conform to the `weblogic-jmsmd.xsd` schema. JMS modules do not include JMS server definitions, which are still stored in the WebLogic domain configuration file.

You create and manage JMS resources either as *system resource modules*, similar to the way they were managed prior to this release, or as *application modules*. JMS application modules are a WebLogic-specific extension of J2EE modules and can be deployed either with a J2EE application (as a packaged resource) or as standalone modules that can be made globally available.

With modular deployment of JMS resources, you can migrate your application and the required JMS configuration from environment to environment, such as from a testing environment to a production environment, without opening an enterprise application file (such as an EAR file) or a standalone JMS module, and without extensive manual JMS reconfiguration.

JMS Servers

JMS servers are environment-related configuration entities that act as management containers for JMS queue and topic resources within JMS modules that are targeted to specific JMS servers. A JMS server's primary responsibility for its targeted destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations. As a container for targeted destinations, any configuration or run-time changes to a JMS server can affect all of its destinations.

As in prior releases, JMS servers are persisted in the domain's `config.xml` file and multiple JMS servers can be configured on the various WebLogic Server instances in a cluster, as long as they are uniquely named. Client applications use either the JNDI tree or the `java:/comp/env` naming context to look up a connection factory and create a connection to establish communication with a JMS server. Each JMS server handles requests for all targeted modules' destinations. Requests for destinations not handled by a JMS server are forwarded to the appropriate server instance.

There are a number of behavioral differences compared to how JMS servers operated in prior releases:

- Because destinations are encapsulated in JMS modules, they are no longer nested under JMS servers in the configuration file. However, the prior "subtargeting" relationship between JMS servers and destinations is still maintained since destination resources within a JMS module are always targeted to JMS servers. This way, JMS servers still manage persistent messages, durable subscribers, message paging, and, optionally, quota for destinations targeted to them. Multiple JMS modules can be targeted to each JMS server in a domain.
- JMS servers support the new Persistent Store that is available to multiple subsystems and services within a server instance, as described in "Persistent Stores" on page 2-12.
  - JMS servers can store persistent messages in a host server's default file store by enabling the "Use the Default Store" option. In prior releases, persistent messages were silently downgraded to non-persistent if no store was configured. Disabling the Use the Default Store option, however, forces persistent messages to be non-persistent.
  - In place of the deprecated JMS stores (JMS file store and JMS JDBC store), JMS servers now support user-defined WebLogic File Stores or JDBC stores, which provide better performance and more capabilities than the legacy JMS stores. (The legacy JMS stores are supported in this release for backward compatibility.)
- JMS servers support an improved message paging mechanism. For more information on message paging, see "Paging Out Messages To Free Up Memory" on page 9-6.
  - The configuration of a dedicated paging store is no longer necessary because paged messages are stored in a directory on your file system -- either to a user-defined directory or to a default paging directory if one is not specified.
  - Temporary paging of messages is always enabled and is controlled by the value set on the Message Buffer Size option. When the total size of non-pending, unpaged messages reaches this setting, a JMS server will attempt to reduce its memory usage by paging out messages to the paging directory.
- You can pause message production or message consumption operations on all the destinations hosted by a single JMS server, either programmatically with JMX or by using the Administration Console. For more information see, "Controlling Message Operations on Destinations" on page 8-15.

- JMS servers can be undeployed and redeployed without having to reboot WebLogic Server.

For more information on configuring JMS servers, see "JMS Server Configuration" on page 3-6.

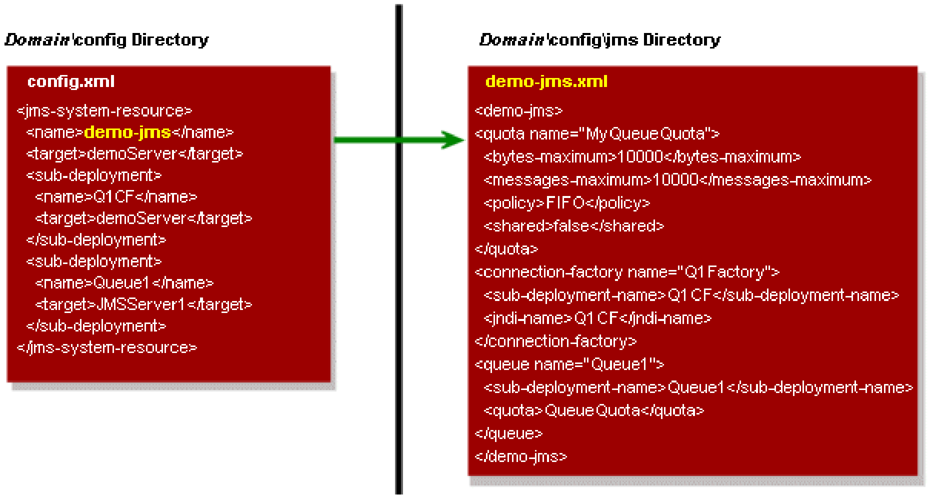
JMS System Resource Modules

When you create a JMS module using the Administration Console, the WebLogic Scripting Tool (WLST), or WebLogic Management Extension (JMX) utilities, the WebLogic Server creates a JMS module file in the `config\jms` subdirectory of the domain directory, and adds a reference to the module in the domain's `config.xml` file as a `JMSSystemResource` element. This reference includes the path to the JMS module file and a list of target servers and clusters on which the module is deployed. The JMS module conforms to the `weblogic-jmsmd.xsd` schema, as described in "JMS Schema" on page 5-2.

JMS resources that you configure this way are considered *system modules*. JMS system modules are owned by the Administrator, who can delete, modify, or add similar resources at any time. System modules are globally available for targeting to servers and clusters configured in the domain, and therefore are available to all applications deployed on the same targets and to client applications. System modules are also accessible through JMX as a `JMSSystemResourceMBean`. The naming convention for JMS system modules is `*MyJMSModule*-jms.xml`.

Figure 2-2 shows an example of a JMS system module listing in the domain's `config.xml` file and the module that it maps to in the `config\jms` directory.

Figure 2-2 Reference from config.xml to a JMS System Module



config.xml

```
<jms-system-resource>
  <name>demo-jms</name>
  <target>demoServer</target>
  <sub-deployment>
    <name>Q1CF</name>
    <target>demoServer</target>
  </sub-deployment>
  <sub-deployment>
    <name>Queue1</name>
    <target>JMSServer1</target>
  </sub-deployment>
</jms-system-resource>
```

demo-jms.xml

```
<demo-jms>
<quota name="MyQueueQuota">
  <bytes-maximum>10000</bytes-maximum>
  <messages-maximum>10000</messages-maximum>
  <policy>FIFO</policy>
  <shared>false</shared>
</quota>
<connection-factory name="Q1Factory">
  <sub-deployment-name>Q1CF</sub-deployment-name>
  <jndi-name>Q1CF</jndi-name>
</connection-factory>
<queue name="Queue1">
  <sub-deployment-name>Queue1</sub-deployment-name>
  <quota>QueueQuota</quota>
</queue>
</demo-jms>
```

The following basic configuration resources are defined as part of a system or packaged JMS module:

- Queue and topic destinations, as described in "Queue and Topic Destination Resources" on page 3-16.
- Connection factories, as described in "Connection Factory Resources" on page 3-13.
- Templates, as described in "JMS Template Resources" on page 3-20.
- Destination keys, as described in "Destination Key Resources" on page 3-22.
- Quota, as described in "Quota Resources" on page 3-23.

The following advanced *clustered* configuration resources are also defined as part of a system or packaged JMS module:

- Distributed destinations, as described in "Configuring Distributed Destinations" on page 4-14.
- Foreign servers, as described in "Accessing Foreign Server Providers" on page 4-11.
- JMS store-and-forward (SAF) configuration items, as described in "Store-and-Forward (SAF) Service" on page 2-12.

All other JMS-related resources must be configured by the administrator as domain configuration resources. This includes:

- JMS servers (required), as described in "JMS Servers" on page 2-5
- Store-and-Forward agents (optional), as described in "Store-and-Forward (SAF) Service" on page 2-12.
- Path service (optional), as described in "Path Service" on page 2-12.
- Messaging bridges (optional), as described in "Messaging Bridges" on page 2-13.
- Persistent stores (optional), as described in "Persistent Stores" on page 2-12

For more information about configuring JMS system modules, see "Configuring JMS System Resources" on page 3-1.

JMS Application Modules

JMS configuration resources can also be managed as deployable application modules, similar to standard J2EE descriptor-based modules. JMS Application modules can be deployed either with a J2EE application as a *packaged module*, where the resources in the module are optionally made available to only the enclosing application (i.e., application-scoped), or as a *standalone module* that provides global access to the resources defined in that module.

Application modules are generally created and packaged by an application developer and are then deployed, managed, and tuned by a WebLogic administrator. As discussed in "Domain Configuration: System Resources vs. Application Resources" on page 2-4, JMS application modules do not contain compiled Java programs as part of the package, enabling administrators or application developers to create and manage JMS resources on demand.

For more information about JMS application modules, see Chapter 5, "Configuring JMS Application Modules for Deployment."

Standalone JMS Modules

A JMS application module can be deployed by itself as a *standalone module*, in which case the module is available to the server or cluster targeted during the deployment process. JMS resources deployed in this manner can be reconfigured using the `weblogic.Deployer` utility or the Administration Console, but are not available through JMX or WLST. However, standalone JMS modules are available using the basic JSR-88 deployment tool provided with WebLogic Server plug-ins (without using WebLogic Server extensions to the API) to configure, deploy, and redeploy J2EE applications and modules to WebLogic Server.

For information about WebLogic Server deployment, see "Understanding WebLogic Server Deployment." For more information about preparing standalone JMS modules for deployment, see "Deploying Standalone JMS Modules" on page 5-9.

Packaged JMS Modules

JMS application modules can also be included as part of an J2EE Enterprise Application Archive (EAR), as a *packaged module*. Packaged modules are bundled with an EAR file or in an exploded EAR directory, and are referenced in the `weblogic-application.xml` deployment descriptor. The packaged module is deployed along with the Enterprise Application, and its resources can be be made available only to the enclosing application. Using packaged modules ensures that an application always has access to required resources and simplifies the process of moving the application into new environments. For more information about preparing packaged JMS modules for deployment, see "Deploying JMS Modules That Are Packaged In an Enterprise Application" on page 5-2.

JMS Schema

In support of the modular configuration model for JMS resources in this release, BEA provides a schema for WebLogic JMS objects: `weblogic-jmsmd.xsd`, where *`jmsmd`* stands for *JMS module descriptor*. When you create JMS resource modules (descriptors), the modules must conform to the schema. IDEs and other tools can validate JMS resource modules based on this schema.

The `weblogic-jmsmd.xsd` schema is available online at `http://www.bea.com/ns/weblogic/90/weblogic-jmsmd.xsd`.

Ownership of Configured JMS Resources

A key to understanding WebLogic JMS configuration and management is that *who* creates a JMS resource and *how* a JMS resource is created determines how a resource is deployed and modified. Both WebLogic administrators and programmers can configure JMS modules:

- WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JMS modules. These JMS modules are considered system resource modules. See "JMS System Resource Modules" on page 2-7 for more details.
- Application developers typically create modules in an enterprise-level IDE or another development tool that supports editing XML descriptor files, then package the JMS modules with an application and pass the application to a WebLogic Administrator to deploy. These JMS modules are considered application modules. See "JMS Application Modules" on page 2-9 for more details.

Comparing System and Application Module Capabilities

In contrast to system modules, deployed application modules are owned by the developer who created and packaged the module, rather than the administrator who deploys the module, which means the administrator has more limited control over deployed resources. When deploying an application module, an administrator can change resource properties that were specified in the module, but the administrator cannot add or delete resources. As with other J2EE modules, deployment configuration changes for a application module are stored in a deployment plan for the module, leaving the original module untouched.

Table 2-1 lists the JMS module types and how they can be configured and modified.

Table 2-1 JMS Module Types and Configuration and Management Options

Module Type	Created with	Dynamically Add/Remove Modules	Modify with JMX Remotely	Modify with Deployment Tuning Plan (non-remote)	Modify with Admin Console	Scoping	Default Sub-module Targeting
System	Admin Console or WLST	Yes	Yes	No	Yes – via JMX	Global and local	No
Packaged	IDE or XML editor	No – must be redeployed	No	Yes – via deployment plan	Yes – via deployment plan	Global, local, and application	Yes

For more information about preparing JMS application modules for deployment, see "Configuring JMS Application Modules for Deployment" on page 5-1and Deploying Applications and Modules in *Deploying Applications to WebLogic Server*

JMS Interop Modules

A JMS interop module is a special type of JMS system resource module. It is created and managed as a result of a JMS configuration upgrade for this release, and/or through the use of WebLogic JMX MBean APIs from prior releases.

JMS interop modules differ in many ways from JMS system resource modules, as follows.

- The JMS module descriptor is always named as `interop-jms.xml` and the file exists in the domain's `config\jms` directory.
- Interop modules are *owned* by the system, as opposed to other JMS system resource modules, which are owned mainly by an administrator.
- Interop modules are targeted everywhere in the domain.
- The JMS resources that exist in a JMS interop module can be accessed and managed using deprecated JMX (MBean) APIs.
- The MBean of a JMS interop module is `JMSInteropModuleMBean`, which is a child MBean of `DomainMBean`, and can be looked up from `DomainMBean` like any other child MBean in a domain.

An interop module can also implement many of the new JMS features in this release, such as "Unit-of-Order" and "Quota". However, it *cannot* implement the following new features:

- Uniform Distributed Destinations
- JMS Store-and Forward

Caution: Use of any new feautures in the current release in a JMS interop module may possibly break compatibility with prior release JMX clients.

Persistent Stores

The WebLogic Persistent Store provides a built-in, high-performance storage solution for all subsystems and services that require persistence. For example, it can store persistent JMS messages or temporarily store messages sent using the Store-and-Forward feature. Each WebLogic Server instance in a domain has a default persistent store that requires no configuration and which can be simultaneously used by subsystems that prefer to use the system's default storage. However, you can also configure a dedicated file-based store or JDBC database-accessible store to suit your JMS implementation. For more information on configuring a persistent store for JMS, see "Using the WebLogic Persistent Store" in *Configuring WebLogic Server Environments*.

Store-and-Forward (SAF) Service

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

JMS modules utilize the SAF service to enable local JMS message producers to reliably send messages to remote JMS queues or topics. For more information, see "Understanding the Store-and-Forward Service" in *Configuring and Managing WebLogic Store-and-Forward*.

Path Service

The WebLogic Server Path Service is a persistent map that can be used to store the mapping of a group of messages to a messaging resource by pinning messages to a distributed queue member or store-and-forward path. For more information on configuring a path service, see *"Using the WebLogic Path Service" on page 4-9*.

Messaging Bridges

The Messaging Bridge allows you to configure a forwarding mechanism between any two messaging products, providing interoperability between separate implementations of WebLogic JMS, or between WebLogic JMS and another messaging product. The messaging bridge instances and bridge source and target destination instances are persisted in the domain's `config.xml` file. For more information, see "Understanding the Messaging Bridge" in *Configuring and Managing WebLogic Messaging Bridge*.

WebLogic Server Value-Added JMS Features

WebLogic JMS provides numerous WebLogic JMS Extension APIs that go above and beyond the standard JMS APIs specified by the JMS 1.1 Specification. Moreover, it is tightly integrated into the WebLogic Server platform, allowing you to build highly-secure J2EE applications that can be easily monitored and administered through the WebLogic Server console. In addition to fully supporting XA transactions, WebLogic JMS also features high availability through its clustering and service migration features, while also providing seamless interoperability with other versions of WebLogic Server and third-party messaging providers.

The following sections provide an overview of the unique features and powerful capabilities of WebLogic JMS.

Note: This section lists only the value-added features for prior releases. For a comprehensive listing of the new WebLogic JMS feature introduced in this release, see "New and Changed JMS Features In This Release" on page 1-4.

Enterprise-Grade Reliability

- *Out-of-the-box transaction support*:
  - Fully supports transactions, including distributed transactions, between JMS applications and other transaction-capable resources using the Java Transaction API (JTA), as described in "Using Transactions with WebLogic JMS" in *Programming WebLogic JMS*.
  - Fully-integrated Transaction Manager, as described in "Introducing Transactions" in *Programming WebLogic JTA*.
- *File or database persistent message storage* (both fully XA transaction capable), as described in "Using the WebLogic Persistent Store" in *Configuring WebLogic Server Environments*.
- *Supports connection clustering* using connection factories targeted on multiple WebLogic Servers, as described in "Configuring WebLogic JMS Clustering" on page 4-2.
- *Distributed destinations* that provide higher destination availability, load balancing, and failover support in a cluster, as described in "Using Distributed Destinations" in *Programming WebLogic JMS*.
- *Failed server migration* for manually restarting JMS servers on another WebLogic Server instance in a cluster, as described in "Configuration Steps for JMS Service Migration" on page 4-7.

- *Redirects failed or expired messages to error destinations*, as described in "Managing Rolled Back, Recovered, Redelivered, or Expired Messages" in *Programming WebLogic JMS*.
- *Provides three levels of load balancing*: network-level, JMS connections, and distributed destinations.

Enterprise-Level Features

- *Message paging* automatically kicks in during peak load periods to free up virtual memory.
- *Message flow control* during peak load periods, including blocking overactive senders, as described in "Controlling the Flow of Messages on JMS Servers and Destinations" on page 9-7 and "Defining Quota" on page 9-1.
- *Timer services* available for scheduled message delivery, as described in "Setting Message Delivery Times" in *Programming WebLogic JMS*.
- *Multicasting of messages* for simultaneous delivery to many clients using IP multicast, as described in "Using Multicasting with WebLogic Server" in *Programming WebLogic JMS*.
- *Flexible expired message policies* to handle expired messages, as described in "Handling Expired Messages" on page 9-11.
- *Supports messages containing XML* (Extensible Markup Language), as described in "Defining XML Message Selectors Using the XML Selector Method" in *Programming WebLogic JMS*.
- *Thin application client* `.JAR` *that provides full WebLogic Server J2EE functionality*, including JMS, yet greatly reduces the client-side WebLogic footprint, as described in "WebLogic JMS Thin Client" in *Programming Stand Alone Clients*.
- *Automatic pooling of JMS client resources in server-side applications* via JMS resource-reference pooling. Server-side applications use standard JMS APIs, but get automatic resource pooling, as described in see "Enhanced 2EE Support for Using WebLogic JMS With EJBs and Servlets" in *Programming WebLogic JMS*.

Tight Integration With WebLogic Server

- *JMS can be accessed locally by server-side applications without a network call* because the destinations can exist on the same server as the application.

- *Uses same ports, protocols, and user identities as WebLogic Server* (T3, IIOP, and HTTP tunnelling protocols, optionally with SSL).
- *Web Services, EJBs, and servlets* supplied by WebLogic Server can work in close concert with JMS.
- *Can be configured and monitored by using the same Administration Console*, or by using the JMS API.
- *Complete JMX administrative and monitoring APIs*, as described in *Developing Custom Management Utilities with JMX*.
- Fully-integrated Transaction Manager, as described in "Introducing Transactions" in *Programming WebLogic JTA*.
- Leverages sophisticated security model built into WebLogic Server (policy engine), as described in the *Understanding WebLogic Security* and "JMS (Java Message Service) Resources" in *Securing WebLogic Resources*.

Interoperability With Other Messaging Services

- *Messages forwarded transactionally by the WebLogic Messaging Bridge* to other JMS providers — as well as to other instances and versions of WebLogic JMS, as described see *Configuring and Managing WebLogic Messaging Bridge*.
- *Supports mapping of other JMS providers* so their objects appear in the WebLogic JNDI tree as local JMS objects. Can also reference remote instances of WebLogic Server in another cluster or domain in the local JNDI tree. For more information, see "Foreign Server Resources" on page 3-23.
- *Uses MDBs to transactionally receive messages* from multiple JMS providers, as described in "Message-Driven EJBs" in *Programming WebLogic Enterprise JavaBeans*.
- *Reliable Web Services integration* with JMS as a transport, as described in "Using Reliable Web Services Messaging" in *Programming WebLogic Web Services*.
- *Automatic transaction enlistment of non-WebLogic JMS client resources* in server-side applications via JMS resource-reference pooling. Server-side applications use standard JMS APIs, but get automatic transaction enlistment, as described in see "Enhanced J2EE Support for Using WebLogic JMS With EJBs and Servlets" in *Programming WebLogic JMS*.

- *Seamless integration with BEA Tuxedo messaging provided by WebLogic Tuxedo Connector* as described in "How to Configure the Tuxedo Queuing Bridge" in the *WebLogic Tuxedo Connector Administration Guide*.

Clustered WebLogic JMS

The WebLogic JMS architecture implements *clustering* of multiple JMS servers by supporting cluster-wide, transparent access to JMS 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 WebLogic Server instances in the cluster.

For more information about configuring clustering for WebLogic JMS, see "Configuring WebLogic JMS Clustering" on page 4-2. For detailed information about WebLogic Server clustering, see *Using WebLogic Server Clusters*.

Configuring JMS System Resources

These sections explain how to configure global JMS system resources for your WebLogic Server implementation:

Ways to Configure Messaging Resources

Messaging resources can be configured in a number of ways:

Administrators

WebLogic Administrators typically use the Administration Console, WebLogic Java Management Extensions (JMX) API, or the WebLogic Scripting Tool (WLST) to create and deploy (target) messaging resources.

- The WebLogic Server Administration Console enables you to configure, modify, and target JMS-related resources:
  - JMS servers, as described in "JMS Server Configuration" on page 3-6.
  - JMS system module resources, as described in "JMS System Module Configuration" on page 3-9.
  - Persistent stores, as described in *Using the WebLogic Persistent Store* in *Configuring WebLogic Server Environments*.
  - Store-and-Forward services for JMS, as described in "*Configuring Store-and-Forward for JMS Messages*" in *Configuring and Managing WebLogic Store-and-Forward*.
- The WebLogic Scripting Tool (WLST) is a command-line scripting interface that allows system administrators and operators to initiate, manage, and persist WebLogic Server configuration changes interactively or by using an executable script. For more information, see Chapter 6, "Using WLST to Manage JMS Servers and JMS System Resources."
- WebLogic Java Management Extensions (JMX) is the J2EE solution for monitoring and managing resources on a network. For more information see "Overview of WebLogic Server Subsystem MBeans" in *Developing Custom Management Utilities with JMX*.

Application Developers

Developers create application modules in an enterprise-level IDE or another development tool that supports editing of XML files, then package the JMS modules with an application and pass the application to a WebLogic Administrator to deploy.

- Configuring JMS application modules, as described in Chapter 5, "Configuring JMS Application Modules for Deployment.".
- Deploying JMS application modules, as described in "Deploying JDBC, JMS, and WLDF Application Modules."

- The JMSModuleHelper extension class contains methods to create and manage JMS module configuration resources in a given module. For more information, see "Using the JMS Module Helper" in *Programming WebLogic JMS* or the JMSModuleHelper Class Javadoc.

Using the Administration Console to Configure JMS Resources

The WebLogic Server Administration Console provides an interface for easily configuring and managing the features of the WebLogic Server, including JMS. To invoke the Administration Console, refer to the procedures described in "Starting and Stopping Servers" in *Managing Server Startup and Shutdown*.

Using the Administration Console, you define configuration options to:

- Configure JMS servers and target them, as described in "JMS Server Configuration" on page 3-6.
- Configure JMS system modules, including queue and topic destinations, connection factories, JMS templates, destination sort keys, destination quota, distributed destinations, foreign servers, and store-and-forward configurations.
- Optionally, create a custom persistent store for JMS, either file-based or a JDBC-accessible table, as described in "Using the WebLogic Persistent Store" in the *Configuring WebLogic Server Environments*.

Modifying Default Values for Configuration Options

WebLogic JMS provides default values for some configuration options; you must provide values for all others. Follow the procedures outlined in this section to configure and manage JMS resources. Once WebLogic JMS is configured, applications can send and receive messages using the JMS API. For more information about developing basis WebLogic JMS applications, refer to "Developing a Basic JMS Application" in *Programming WebLogic JMS*.

A sample `examplesJMSServer` configuration is provided with the product in the Examples Server. For more information about starting the Examples Server, see "Starting and Stopping Servers" in *Managing Server Startup and Shutdown*.

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 WebLogic 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 Server instances, 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 *Managing Server Startup and Shutdown*.

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 instructions about using the Administration Console to manage a WebLogic Server domain, see "The WebLogic Server Administration Console" in the *Administration Console Online Help*.

Configuring Basic JMS System Resources

This section describes how to configure a persistent store, a JMS server, and a basic JMS system module using the Administration Console.

This section does not cover the configuration parameters available to fine-tune JMS resources once they are created. For information about these parameters, refer to the corresponding system module beans in the "System Module MBeans" folder of the *WebLogic Server MBean Reference*. The root bean in the JMS module that represents an entire JMS module is named JMSBean.

1. For storing persistent messages in a file-based store, you can simply use the server's default persistent store, which requires no configuration on your part. However, you can also create a dedicated file store for JMS.

   For more information on configuring a custom file store, see "Creating a Custom (User-Defined) File Store" in the *Configuring WebLogic Server Environments*.

2. For storing persistent messages in a JDBC-accessible database, you must create a JDBC store.

   For more information on JDBC stores, see "Creating a JDBC Store" in *Configuring WebLogic Server Environments*.

3. Create a JMS server to manage the queue and topic destinations in a JMS system module.

   For more information, see "JMS Servers" on page 2-5.

4. Create a JMS system module to contain your JMS system resources, such as quota, templates, destination keys, standalone queue and topic destinations, and connection factories.

   For more information, see "JMS System Resource Modules" on page 2-7.

5. Before creating any queue or topic resources in your system module, you can optionally create other JMS resources in the module that can be referenced from within a queue or topic, such as JMS templates, quota settings, and destination sort keys:

   – Define quota resources and then have destinations refer to them. Destinations can be assigned their own quotas; multiple destinations can share a quota; or destinations can share the JMS server's quota.

   – Create JMS templates, which allow you to define multiple destinations with similar option settings. For more information, see "JMS Template Resources" on page 3-20.

   – Configure destination keys to create custom sort orders of messages as they arrive on a destination. For more information, see "Destination Key Resources" on page 3-22.

   Once these resources are configured, they can be selected when configuring your queue or topic resources.

6. Create a topic to include in your JMS module.

   For more information on configuring JMS topics, see "Creating Topics" on page 3-18.

7. Create a queue to include in your JMS module.

   For more information on configuring JMS queues, see "Creating Queues" on page 3-17.

8. If the default connection factories provided by WebLogic Server are not suitable for your application, create a connection factory to enable your JMS clients to create JMS connections.

   For more information about using the default connection factories, see "Using a Default Connection Factory" on page 3-13. For more information on configuring a Connection Factory, see "Creating Connection Factories" on page 3-14.

Guidelines for Configuring Advanced JMS System Module Resources

This section describes the advanced resources that can be added to a JMS system module using the Administration Console.

- Create a Distributed Destination to make your physical destinations part of a single distributed destination set within a server cluster. For more information, see "Configuring Distributed Destinations" on page 4-14.

- Create a Store-and-Forward configuration to reliably forward messages to remote destinations, even if a destination is unavailable at the time a message is sent, as described in *Configuring and Managing WebLogic Store-and-Forward*.
- Create a foreign server to reference foreign JMS providers within a local WebLogic Server JNDI tree. For more information, see "Accessing Foreign Server Providers" on page 4-11.

JMS Configuration Naming Requirements

Each server instance within a domain must have a name that is unique for all configuration objects in the domain. Within a domain, each server, machine, cluster, virtual host, and any other resource type must be named uniquely and must not use the same name as the domain. This unique naming rule also applies to all configurable JMS objects, such as JMS servers, JMS system modules, and JMS application modules.

The resource names inside JMS modules must be unique per resource type (e.g., queues, topics, or connection factories). However, two different JMS modules can have a resource of the same type that can share the same name.

Also, the JNDI name of any bindable JMS resource (excluding quotas, destination keys, and JMS templates) across JMS modules has to be unique.

JMS Server Configuration

JMS servers are environment-related configuration entities that act as management containers for JMS queue and topic resources within JMS modules that are specifically targeted to JMS servers. A JMS server's primary responsibility for its targeted destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations. As a container for targeted destinations, any configuration or run-time changes to a JMS server can affect all of its destinations.

There are a number of ways to create JMS servers.

- The WebLogic Server Administration Console enables you to configure, modify, target, monitor, and delete JMS servers in your environment. For a road map of the JMS server configuration tasks, see "Configure JMS servers" in the *Administration Console Online Help*.
- The WebLogic Scripting Tool (WLST) enables you to create JMS servers. For more information, see Chapter 6, "Using WLST to Manage JMS Servers and JMS System Resources."

- WebLogic Java Management Extensions (JMX) enables you to access the `JMSServerMBean` and `JMSServerRuntimeMBean` MBeans to create and manage JMS servers. For more information see "Overview of WebLogic Server Subsystem MBeans" in *Developing Custom Management Utilities with JMX*.
- JMS Module Helper Extension APIs enable you to manage (locate/create/delete) JMS servers. For more information, see "Using the JMS Module Helper" in *Programming WebLogic JMS* or the `JMSModuleHelper` Javadoc.

Creating JMS Servers

As management containers for destinations, JMS servers feature the following configuration parameters:

- General configuration parameters, including selecting a persistent storage, setting message paging defaults, specifying a template to use when your applications create temporary destinations, and specifying expired message scanning parameters.
- Threshold and quota parameters for destinations in JMS modules targeted to this JMS server.

  For more information about configuring messages and bytes quota for JMS servers and destinations, see "Defining Quota" on page 9-1.
- Message logging parameters for a JMS server's log file, which contains the basic events that a JMS message traverses through, such as message production, consumption, and removal.

  For more information about configuring message life cycle logging on JMS servers, see "Message Life Cycle Logging" on page 8-7.
- Destination pause/resume controls that enable you to pause message production, message insertion (in-flight messages), and message consumption operations on all the destinations hosted by a single JMS Server.

  For more information about pausing message operations on destinations, see "Controlling Message Operations on Destinations" on page 8-15.

Some JMS server options are dynamically configurable. When options are modified at runtime, only incoming messages are affected; stored messages are not affected. For more information about the default values for all JMS server options, see `JMSServerBean` in the *WebLogic Server MBean Reference*.

Targeting JMS Servers

You can target a JMS server to either an independent WebLogic Server instance or to a migratable target server where it will be deployed.

- Weblogic Server instance — the server target where you want to deploy the JMS server. When a target WebLogic Server boots, the JMS server boots as well. If no target WebLogic Server is specified, the JMS server will not boot. The deployment of a JMS server differs from that of a connection factory, as described in "Targeting Connection Factories" on page 3-14.
- Migratable Target — the migratable target where you want to deploy the JMS server. Migratable targets define a set of WebLogic Server instances in a cluster that can potentially host an *exactly-once* service, such as JMS. 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 server failure or due to a scheduled migration for system maintenance.

For more information on configuring a migratable target for JMS, see "Configuring Migratable Targets for JMS Servers" on page 4-7.

Monitoring JMS Servers

You can monitor run-time statistics for active JMS servers, destinations, and server session pools.

- Monitor all Active JMS Servers — A table displays showing all instances of the JMS server deployed across the WebLogic Server domain.
- Monitor all Active JMS Destinations — A table displays showing all active JMS destinations for the current domain.
- Monitor all Active JMS Session Pool Runtimes — A table displays showing all active JMS session pools for the current domain.

For more information about monitoring JMS objects, see "Monitoring JMS Statistics and Managing Messages" on page 7-1.

Configuring Session Pools and Connection Consumers

Note: Session pool and connection consumer configuration objects are deprecated in this release of WebLogic Server. They are not a required part of the J2EE specification, do not support JTA user transactions, and are largely superseded by Message-Driven Beans

(MDBs), which are a required part of J2EE. For more information on designing MDBs, see "Message-Driven EJBs" in *Programming WebLogic Enterprise JavaBeans*.

Server session pools enable an application to process messages concurrently. After you define a JMS server, you can configure one or more session pools for each JMS server. Some session pool options are dynamically configurable, but the new values do not take effect until the JMS server is restarted. For more information about creating session pools, see Defining Server Session Pools in *Programming WebLogic JMS*.

Connection consumers are queues (Point-To-Point) or topics (Pub/Sub) that will retrieve server sessions and process messages. After you define a session pool, configure one or more connection consumers for each session pool. For more information about creating connection consumers, see "Defining Server Session Pools in *Programming WebLogic JMS*.

JMS System Module Configuration

JMS system modules are owned by the Administrator, who can delete, modify, or add JMS system resources at any time. With the exception of standalone queue and topic resources that must be targeted to a single JMS server, the connection factory, distributed destination, foreign server, and JMS SAF destination resources in system modules can be made globally available by targeting them to server instances and clusters configured in the WebLogic domain. These resources are therefore available to all applications deployed on the same targets and to client applications. The naming convention for JMS system modules is `*MyJMSModule*-jms.xml`.

The following basic configuration resources are defined as part of a JMS system module:

- Queue and topic destinations, as described in "Queue and Topic Destination Resources" on page 3-16.
- Connection factories, as described in "Connection Factory Resources" on page 3-13.
- Templates, as described in "JMS Template Resources" on page 3-20.
- Destination keys, as described in "Destination Key Resources" on page 3-22.
- Quota, as described in "Quota Resources" on page 3-23.

The following "advanced" clustering configuration resources are also defined as part of a JMS system module:

- Foreign servers, as described in "Accessing Foreign Server Providers" on page 4-11.
- Distributed destinations, as described in "Configuring Distributed Destinations" on page 4-14.

- JMS store-and-forward configurations, as described in "Configuring SAF for JMS Messages" in *Configuring and Managing WebLogic Store-and-Forward*.

Creating JMS System Modules

There are a number of ways to create a JMS system module and its resources.

- The WebLogic Server Administration Console enables you to configure, modify, target, monitor, and delete JMS system modules in your environment. For a road map of the JMS system module configuration tasks, see "Configure JMS system modules and add JMS resources" in the *Administration Console Online Help*.
- The WebLogic Scripting Tool (WLST) enables you to create JMS servers. For more information, see Chapter 6, "Using WLST to Manage JMS Servers and JMS System Resources."
- WebLogic Java Management Extensions (JMX) enables you to access the `JMSSystemResourceMBean` and `JMSRuntimeMBean` MBeans to create and manage JMS destinations and connections. For more information see "Overview of WebLogic Server Subsystem MBeans" in *Developing Custom Management Utilities with JMX*.
- JMS Module Helper Extension APIs enable you to locate JMS runtime MBeans, as well as methods to manage (locate/create/delete) JMS system module configuration resources in a given module. For more information, see "Using the JMS Module Helper" in *Programming WebLogic JMS* or the `JMSModuleHelper` Javadoc.

Targeting JMS Modules and Subdeployment Resources

JMS system modules must be targeted to one or more WebLogic Server instances or to a cluster. Targetable resources defined in a system module must also be targeted to JMS server or WebLogic Server instances within the scope of a parent module's targets. Additionally, targetable JMS resources inside a system module can be further grouped into *subdeployments* during the configuration or targeting process to provide further loose coupling of JMS resources in a WebLogic domain.

A subdeployment is a mechanism by which targetable JMS system module resources (such as queues, topics, and connection factories) are grouped and targeted to specific server resources within a system module's targeting scope. Although a JMS system module can be targeted to a wide array of WebLogic Server instances in a domain, a module's standalone queues or topics can only be targeted to a single JMS server. Whereas, connection factories, uniform distributed destinations (UDDs), and foreign servers can be targeted to one or more JMS servers, one or more WebLogic Server instances, or to a cluster. Therefore, standalone queues or topics cannot be associated with a subdeployment if other members of the subdeployment are targeted to multiple JMS servers. However, UDDs can be associated with such subdeployments since the purpose of UDDs is to distribute its members to multiple JMS servers in a domain.

Table 3-1 shows the valid targeting options for JMS system resource subdeployments:

Table 3-1 JMS System Resource Subdeployment Targeting

JMS Resource	Valid Targets
Queue	JMS server
Topic	JMS server
Connection factory	JMS server(s) | server instance(s) | cluster
Distributed queue	JMS server(s) | server instance(s) | cluster
Distributed topic	JMS server(s) | server instance(s) | cluster
Foreign server	JMS server(s) | server instance(s) | cluster
SAF Imported Destinations	SAF Agent(s) | server instance(s) | cluster

An example of a simple subdeployment for standalone queues or topics would be to group them with a connection factory so that these resources are co-located on a specific JMS server, which can help reduce network traffic. Also, if the targeted JMS server should be migrated to another WebLogic Server instance, the connection factory and all its connections will also migrate along with the JMS server's destinations.

For example, if a system module named *jmssysmod-jms.xml*, is targeted to a WebLogic Server instance that has two configured JMS servers: *jmsserver1* and *jmsserver2*, and you want to co-locate two queues and a connection factory on only *jmsserver1*, you can group the queues and connection factory in the same subdeployment, named *jmsserver1group*, to ensure that these resources are always linked to *jmsserver1*, provided the connection factory is not already targeted to multiple JMS servers.

```
<weblogic-jms xmlns="http://www.bea.com/ns/weblogic/90">
  <connection-factory name="connfactory1">
    <sub-deployment-name>jmsserver1group</sub-deployment-name>
    <jndi-name>cf1</jndi-name>
  </connection-factory>
 <queue name="queue1">
    <sub-deployment-name>jmsserver1group</sub-deployment-name>
    <jndi-name>q1</jndi-name>
  </queue>
 <queue name="queue2">
    <sub-deployment-name>jmsserver1group</sub-deployment-name>
    <jndi-name>q2</jndi-name>
  </queue>
</weblogic-jms>
```

And here's how the *jmsserver1group* subdeployment targeting would look in the domain's configuration file:

```
  <jms-system-resource>
   <name>jmssysmod-jms</name>
   <target>wlsserver1</target>
   <sub-deployment>
     <name>jmsserver1group</name>
     <target>jmsserver1</target>
   </sub-deployment>
   <descriptor-file-name>jms/jmssysmod-jms.xml</descriptor-file-name>
  </jms-system-resource>
```

For information about deploying stand-alone JMS modules, see "Deploying JDBC and JMS Application Modules."

Connection Factory Resources

Connection factories are objects that enable JMS clients to create JMS connections. A connection factory supports concurrent use, enabling multiple threads to access the object simultaneously. WebLogic JMS provides pre-configured "default connection factories" that can be enabled or disabled on a per-server basis, as described in "Using a Default Connection Factory" on page 3-13.

Otherwise, you can configure one or more connection factories to create connections with predefined options that better suit your application. Within each JMS module, connection factory resource names must be unique. However, all connection factory JNDI names in any JMS module must be unique across an entire WebLogic domain, as defined in "JMS Configuration Naming Requirements" on page 3-6. WebLogic Server adds them to the JNDI space during startup, and the application then retrieves a connection factory using the WebLogic JNDI APIs.

You can establish cluster-wide, transparent access to JMS destinations from any server in the cluster, either by using the default connection factories for each server instance, or by configuring one or more connection factories and targeting them to one or more server instances in the cluster. This way, each connection factory can be deployed on multiple WebLogic Servers. For more information on configuring JMS clustering, see "Configuring WebLogic JMS Clustering" on page 4-2.

Using a Default Connection Factory

WebLogic JMS defines two default connection factories, which can be looked up using the following JNDI names:

- `weblogic.jms.ConnectionFactory`
- `weblogic.jms.XAConnectionFactory`

You only need to configure a new connection factory if the pre-configured settings of the default factories are not suitable for your application. For more information on using the default connection factories, see "Understanding WebLogic JMS" in *Programming WebLogic JMS*

The main difference between the pre-configured settings for the default connection factories and a user-defined connection factory is the default value for the "XA Connection Factory Enabled" option to enable JTA transactions. For more information about the XA Connection Factory

Enabled option, and to see the default values for the other connection factory options, see `JMSConnectionFactoryBean` in the *WebLogic Server MBean Reference*.

Another distinction when using the default connection factories is that you have no control over targeting the WebLogic Server instances where the connection factory may be deployed. However, you can enable and/or disable the default connection factories on a per-WebLogic Server basis, as defined in "Server: Services: Configuration" in the *Administration Console Online Help*.

Creating Connection Factories

The WebLogic Server Administration Console enables you to configure, modify, target, and delete connection factory resources in a system module. For a road map of the JMS connection configuration tasks, see "Configure connection factories" in the *Administration Console Online Help*.

Connection factories feature the following configuration parameters:

- General configuration parameters, including modifying the default client parameters, default message delivery parameters, load balancing parameters, unit-of-order parameters, and security parameters.
- Transaction parameters, which enable you to define a value for the transaction time-out option and to indicate whether an XA queue or XA topic connection factory is returned, and whether the connection factory creates sessions that are JTA aware.
- Flow control parameters, which enable you to tell a JMS server or destination to slow down message producers when it determines that it is becoming overloaded.

Some connection factory options are dynamically configurable. When options are modified at runtime, only incoming messages are affected; stored messages are not affected. For more information about the default values for all connection factory options, see `JMSConnectionFactoryBean` in the *WebLogic Server MBean Reference*.

Targeting Connection Factories

You can target connection factories to one or more JMS server, to one or more WebLogic Server instances, or to a cluster.

- JMS server(s) — You can target connection factories to one or more JMS servers along with destinations. You can also group a connection factory with standalone queues or topics in a subdeployment targeted to a specific JMS server, which guarantees that all these resources are co-located to avoid extra network traffic. Another advantage of such a configuration would be if the targeted JMS server needs to be migrated to another WebLogic server instance, then the connection factory and all its connections will also migrate along with the JMS server's destinations. However, when standalone queues or topics are members of a subdeployment, a connection factory can only be targeted to the same JMS server.

- Weblogic server instance(s) — To establish transparent access to JMS destinations from any server in a domain, you can target a connection factory to multiple WebLogic Server instances simultaneously.
- Cluster — To establish cluster-wide, transparent access to JMS destinations from any server in a cluster, you can target a connection factory to all server instances in the cluster, or even to specific servers within the cluster.

For more information on JMS system module subdeployment targeting, see "Targeting JMS Modules and Subdeployment Resources" on page 3-11.

Queue and Topic Destination Resources

A JMS destination identifies a queue (point-to-point) or topic (publish/subscribe) resource within a JMS module. Each queue and topic resource is targeted to a specific JMS server. A JMS server's primary responsibility for its targeted destinations is to maintain information on what persistent store is used for any persistent messages that arrive on the destinations, and to maintain the states of durable subscribers created on the destinations.

You can optionally create other JMS resources in a module that can be referenced from within a queue or topic, such as JMS templates, quota settings, and destination sort keys:

- Quota — Destinations can be assigned their own quotas; multiple destinations can share a quota; or destinations can share the JMS server's quota. For more information, see "Defining Quota" on page 9-1.
- JMS Template — allows you to define multiple destinations with similar option settings. You also need a JMS template to create temporary queues. For more information, see "JMS Template Resources" on page 3-20.
- Destination Key — create custom sort orders of messages as they arrive on a destination. For more information, see "Destination Key Resources" on page 3-22.

Error Destinations

To help manage recovered or rolled back messages, you can also configure a target error destination for messages that have reached their redelivery limit. The error destination can be either a topic or a queue, but it must be a destination that is targeted to same JMS server as the destination(s) it is associated with. For more information, see "Configuring an Error Destination for Undelivered Messages" in *Programming WebLogic JMS*.

Note: Because error destinations must be targeted to the same JMS server as the destination(s) it is associated with, error destinations cannot be used with distributed destinations since distributed destination members are targeted to multiple JMS servers.

Distributed Destinations

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

Weblogic Server supports two types of distributed destinations:

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

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

For more information on configuring distributed destination resources, see "Configuring Distributed Destinations" on page 4-14.

Creating Queues

A JMS queue defines a *point-to-point* destination type for a JMS server. Queues are used for asynchronous peer communications.A message delivered to a queue will be distributed to one consumer.

The WebLogic Server Administration Console enables you to configure, modify, target, and delete queue resources in a system module. For a road map of the queue tasks, see "Configure queues" in the *Administration Console Online Help*. Within each JMS module, queue resource names must be unique. However, all queue JNDI names in any JMS module must be unique across an entire WebLogic domain, as defined in "JMS Configuration Naming Requirements" on page 3-6.

Queues feature the following configuration parameters:

- General configuration parameters, including specifying a JNDI name, selecting a destination key for sorting messages as they arrive on the queue, or selecting a JMS template if you are using one to configure properties for multiple queues.

  Note: Although queue JNDI names can be dynamically changed, there may be long-lived producers or consumers, such as MDBs, that will continue trying to produce/consume messages to/from the queue's original JNDI name.

- Threshold and quota parameters, for defining the upper and lower message/byte threshold and maximum quota options for the queue.

  For more information about configuring message and bytes quota resources for queues, see "Quota Resources" on page 3-23.

- Message logging parameters, for enabling the logging of message life cycle information into a JMS log file and configuring information like message type and user properties.

For more information about configuring message life cycle logging on queues, see "Message Life Cycle Logging" on page 8-7.

- Pause/resume controls that enable you to pause/resume message production, message insertion (in-flight messages), and message consumption operations on a queue.

  For more information about pausing message operations on queues, see "Controlling Message Operations on Destinations" on page 8-15.

- Message delivery override parameters, such as message priority and time-to-deliver values, that can override those specified by a message producer or connection factory.
- Message Delivery failure parameters, such as defining a message redelivery limit, selecting a message expiration policy, and specifying an error destination for expired messages.

Some queue options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected. For more information about the default values for all queue options, see `QueueBean` in the *WebLogic Server MBean Reference*.

Creating Topics

A JMS topic identifies a *publish/subscribe* destination type for a JMS server. Topics are used for asynchronous peer communications. A message delivered to a topic will be distributed to all topic consumers.

The WebLogic Server Administration Console enables you to configure, modify, target, and delete topic resources in a system module. For a road map of the topic tasks, see "Configure topics" in the *Administration Console Online Help*. Within each JMS module, topic resource names must be unique. However, all topic JNDI names in any JMS module must be unique across an entire WebLogic domain, as defined in "JMS Configuration Naming Requirements" on page 3-6.

Topics feature the following configuration parameters:

- General configuration parameters, including specifying a JNDI name, selecting a destination key for sorting messages as they arrive on the topic, or selecting a JMS template if you are using one to configure properties for multiple topics.

  Note: Although topic JNDI names can be dynamically changed, there may be long-lived producers or consumers, such as MDBs, that will continue trying to produce/consume messages to/from the topic's original JNDI name.

- Threshold and quota parameters, for defining the upper and lower message/byte threshold and maximum quota options for the topic.

  For more information about configuring message and bytes quota resources for topics, see "Quota Resources" on page 3-23.

- Message logging parameters, for enabling the logging of message life cycle information into a JMS log file and configuring information like message type and user properties.

  For more information about configuring message life cycle logging on topics, see "Message Life Cycle Logging" on page 8-7.

- Pause/resume controls that enable you to pause message production, message insertion (in-flight messages), and message consumption operations on a topic.

  For more information about pausing message operations on topics, see "Controlling Message Operations on Destinations" on page 8-15.

- Message delivery override parameters, such as message priority and time-to-deliver values, that can override those specified by a message producer.

- Message Delivery failure parameters, such as defining a message redelivery limit, selecting a message expiration policy, and specifying an error destination for expired messages.

- Multicast parameters, define the multicast options for the topic, including a multicast address, time-to-live (TTL), and port.

Some topic options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected. For more information about the default values for all topic options, see `TopicBean` in the *WebLogic Server MBean Reference*.

Targeting Queues and Topics

Standalone queues and topics can only be deployed to a specific JMS server in a domain because they depend on the JMS servers they are targeted to for the management of persistent messages, durable subscribers, and message paging.

If you want to associate a group of queues and/or topics with a connection factory on a specific JMS server, you can target the destinations and connection factory to the same subdeployment, which links these resources to the JMS server targeted by the subdeployment. However, when standalone destinations are members of a subdeployment, a connection factory can only be targeted to the same JMS server.

For more information on JMS system module subdeployment targeting, see "Targeting JMS Modules and Subdeployment Resources" on page 3-11.

JMS Template Resources

A JMS template provides an efficient means of defining multiple destinations with similar option settings. JMS templates offer the following benefits:

- You do not need to re-enter every option setting each time you define a new destination; you can use the JMS template and override any setting to which you want to assign a new value.
- You can modify shared option settings dynamically simply by modifying the template.
- You can specify subdeployments for error destinations so that any number of destination subdeployments (groups of queue or topics) will use only the error destinations specified in the corresponding template subdeployments.

The configurable options for a JMS template are the same as those configured for a destination. These configuration options are inherited by the destinations that use them, with the following exceptions:

- If the destination that is using a JMS template specifies an override value for an option, the override value is used.
- If the destination that is using a JMS template specifies a message redelivery value for an option, that redelivery value is used.
- The Name option is not inherited by the destination. This name is valid for the JMS template only. You must explicitly define a unique name for all destinations. For more information, see "JMS Configuration Naming Requirements" on page 3-6.
- The JNDI Name, Enable Store, and Template options are not defined for JMS templates.

Any options that are not explicitly defined for a destination are assigned default values. If no default value exists, be sure to specify a value within the JMS template or as a destination option override.

Creating JMS Templates

The WebLogic Server Administration Console enables you to configure, modify, target, and delete JMS template resources in a system module. For a road map of the JMS template tasks, see "Configure JMS templates" in the *Administration Console Online Help*.

JMS templates feature the following configuration parameters for destinations that use the template:

- General configuration parameters, including selecting a destination key for sorting messages as they arrive on destinations.
- Threshold and quota parameters, for defining the upper and lower message/byte threshold and maximum quota options for destinations.

  For more information about configuring message and bytes quota resources for destinations, see "Quota Resources" on page 3-23.
- Message logging parameters, for enabling the logging of message life cycle information into a JMS log file and configuring information like message type and user properties.

  For more information about configuring message life cycle logging on destinations, see "Message Life Cycle Logging" on page 8-7.
- Pause/resume controls that enable you to pause/resume message production, message insertion (in-flight messages), and message consumption operations on a topic.

  For more information about pausing message operations on topics, see "Controlling Message Operations on Destinations" on page 8-15.
- Message delivery override parameters, such as message priority and time-to-deliver values, that can override those specified by a message producer.
- Message Delivery failure parameters, such as defining a message redelivery limit, selecting a message expiration policy, and specifying an error destination for expired messages.
- Multicast parameters, define the multicast options for the topic, including a multicast address, time-to-live (TTL), and port.
- Subdeployments for error destinations, so that any number of destination subdeployments (groups of queue or topics) will use only the error destinations specified in the corresponding template subdeployments.

Some template options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected. For more information about the default values for all topic options, see `TemplateBean` in the *WebLogic Server MBean Reference*.

Destination Key Resources

As messages arrive on a specific destination, by default they are sorted in FIFO (first-in, first-out) order, which sorts ascending based on each message's unique JMSMessageID. However, you can use a destination key to configure a different sorting scheme for a destination, such as LIFO (last-in, first-out).

Creating JMS Destination Keys

The WebLogic Server Administration Console enables you to configure, modify, target, and delete destination key resources in a system module. For a road map of the destination key tasks, see "Configure destination keys" in the *Administration Console Online Help*.

For more information about the default values for all destination key options, see `DestinationKeyBean` in the *WebLogic Server MBean Reference*.

Quota Resources

In prior releases, there were multiple levels of quotas: destinations had their own quotas and would also have to compete for quota within a JMS server. In this release, there is only one level of quota: destinations can have their own private quota resource or they can compete with other destinations using a shared quota resource.

For more information on configuring quota resources, see "Defining Quota" on page 9-1.

Foreign Server Resources

Foreign server resources enable you to reference foreign (that is, third-party) JMS providers within a local WebLogic Server JNDI tree. With a foreign server resource, you can quickly map a foreign JMS provider so that its associated connection factories and destinations appear in the WebLogic JNDI tree as local JMS objects. A Foreign Server resource can also be used to reference remote instances of WebLogic Server in another cluster or domain in the local WebLogic JNDI tree.

For more information on configuring foreign servers resources, see "Accessing Foreign Server Providers" on page 4-11.

Distributed Destination Resources

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

Weblogic Server supports two types of distributed destinations:

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

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

For more information on configuring distributed destination resources, see "Configuring Distributed Destinations" on page 4-14.

JMS Store-and-Forward (SAF) Resources

JMS SAF resources build on the WebLogic Store-and-Forward (SAF) service to provide highly-available JMS message production. For example, a JMS message producer connected to a local server instance can reliably forward messages to a remote JMS destination, even though that remote destination may be temporarily unavailable when the message was sent. JMS Store-and-forward is transparent to JMS applications; therefore, JMS client code still uses the existing JMS APIs to access remote destinations

For more information on configuring JMS SAF resources, see "Configuring SAF for JMS Messages" in *Configuring and Managing WebLogic Store-and-Forward*.

Configuring Clustered WebLogic JMS Resources

These sections provide information on configuring WebLogic JMS resources in a clustered environment:

Configuring WebLogic JMS Clustering

A WebLogic Server *cluster* is a group of servers in a domain 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.

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.

Advantages of JMS Clustering

The advantages of clustering for JMS include the following:

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

  An administrator can establish load balancing of destinations across multiple servers in the cluster by configuring multiple JMS servers and targeting them to the defined WebLogic Servers. Each JMS server is deployed on exactly one WebLogic Server instance 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.

- *High availability of destinations*
  - *Distributed destinations* — The queue and topic members of a distributed destination are usually distributed across multiple servers within a cluster, with each member belonging to a separate JMS server. Applications that use distributed destinations are more highly available than applications that use simple destinations because WebLogic JMS provides load balancing and failover for member destinations of a distributed destination within a cluster. For more information on distributed destinations, see "Configuring Distributed Destinations" on page 4-14.
  - *Store-and-Forward* — JMS modules utilize the SAF service to enable local JMS message producers to reliably send messages to remote queues or topics. If the destination is not available at the moment the messages are sent, either because of network problems or system failures, then the messages are saved on a local server instance, and are forwarded to the remote destination once it becomes available. For more information, see "Understanding the Store-and-Forward Service" in *Configuring and Managing WebLogic Store-and-Forward.*
  - For automatic failover, WebLogic Server supports migration at the server level—a complete server instance, and all of the services it hosts can be migrated to another

machine, either automatically, or manually. For more information, see "Server Migration" in *Using WebLogic Server Clusters*.

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

  An administrator can establish cluster-wide, transparent access to destinations from any server in the cluster by either using the default connection factories for each server instance in the cluster, or by configuring one or more connection factories and targeting them to one or more server instances in the cluster, or to the entire cluster. This way, each connection factory can be deployed on multiple WebLogic Server instances. Connection factories are described in more detail in "Connection Factory Resources" on page 3-13.

- *Scalability*

  – Load balancing of destinations across multiple servers in the cluster, as described previously.

  – Distribution of application load across multiple JMS servers through 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*

  WebLogic Server supports migration at the server level—a complete server instance, and all of the services it hosts can be migrated to another machine, either automatically, or manually. For more information, see "Server Migration" in *Using WebLogic Server Clusters*.

  Also, as an "exactly-once" service, WebLogic JMS takes advantage of the service migration framework implemented in WebLogic Server for clustered environments. This allows WebLogic JMS to respond properly to migration requests and to 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 Migratable Targets for JMS Servers" on page 4-7.

- *Server affinity for JMS Clients*

  When configured for the cluster, load balancing algorithms (round-robin-affinity, weight-based-affinity, or random-affinity), provide server affinity for JMS client connections. If a JMS application has a connection to a given server instance, JMS attempts to establish new JMS connections to the same server instance. For more information on server affinity, see "Load Balancing in a Cluster" in *Using WebLogic Server Clusters*.

For more information about the features and benefits of using WebLogic clusters, see "Understanding WebLogic Server Clustering" in *Using WebLogic Server Clusters*.

Obtain a Clustered JMS Licence

In order to implement JMS clustering, you must have a valid clustered JMS license, which allows a connection factory and a destination to be targeted to different WebLogic Server instances. A clustered JMS license is also required to use:

- Foreign servers, as described in "Accessing Foreign Server Providers" on page 4-11.
- Distributed destinations across multiple WebLogic Server instances, as described in "Configuring Distributed Destinations" on page 4-14.

If you do not have a valid clustered JMS license, contact your BEA sales representative.

How JMS Clustering Works

An administrator can establish cluster-wide, transparent access to JMS destinations from any server in a cluster, either by using the default connection factories for each server instance in a cluster, or by configuring one or more connection factories and targeting them to one or more server instances in a cluster, or to an entire cluster. This way, each connection factory can be deployed on multiple WebLogic Servers. For information on configuring and deploying connection factories, see "Creating Connection Factories" on page 3-14.

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. If requests for destinations are sent to a WebLogic Server instance that is hosting a connection factory, but which is not hosting a JMS server or destinations, the requests are forwarded by the connection factory to the appropriate WebLogic Server instance that is hosting the JMS server and destinations.

The administrator can also configure multiple JMS servers on the various servers in the cluster—as long as the JMS servers are uniquely named—and can then target JMS queue or topic resources 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 instance. For information on configuring and deploying JMS servers, see "JMS Server Configuration" on page 3-6.

JMS Clustering Naming Requirements

There are naming requirements when configuring JMS objects and resources, such as JMS servers, JMS modules, and JMS resources, to work in a clustered environment in a single WebLogic domain or in a multi-domain environment. For more information, see "JMS Configuration Naming Requirements" on page 3-6.

Distributed Destination Within a Cluster

A distributed destination resource is a single set of destinations (queues or topics) that are accessible as a single, logical destination to a client (for example, a distributed topic has its own JNDI name). The members of the unit are usually distributed across multiple servers within a cluster, with each member belonging to a separate JMS server. Applications that use distributed destinations are more highly available than applications that use simple destinations because WebLogic Server provides load balancing and failover for member destinations of a distributed destination within a cluster. For more information, see "Configuring Distributed Destinations" on page 4-14.

JMS As a Migratable Service Within a Cluster

In addition to being part of a whole server migration, where all services hosted by a server can be migrated to another machine, WebLogic JMS is also part of the service migration framework that allows an administrator to migrate a JMS server and all of its destinations can migrate to another WebLogic Server within a cluster. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure. For more information on JMS service migration, see "Configuring Migratable Targets for JMS Servers" on page 4-7.

Configuration Guidelines for JMS Clustering

In order to use WebLogic JMS in a clustered environment, follow these guidelines:

1. Configure your clustered environment as described in "Setting Up WebLogic Clusters" in *Using WebLogic Server Clusters*.
2. Identify server targets for any user-defined JMS connection factories using the *Administration Console*. For connection factories, you can identify either a single-server target or a cluster target, which are server instances that are associated with a connection factory to support clustering.

For more information about these connection factory configuration attributes, see "Connection Factory Resources" on page 3-13.

3. Optionally, identify migratable server targets for JMS servers using the *Administration Console*. For JMS servers, you can identify either a single-server target or a migratable target, which is a set of server instances in a cluster that can host an "exactly-once" service like JMS in case of a server failure in the cluster.

   For more information on migratable JMS server targets, see "Configuring Migratable Targets for JMS Servers" on page 4-7. For more information about JMS server configuration attributes, see "JMS Server Configuration" on page 3-6.

   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.

4. Optionally, you can configure the physical JMS destinations in a cluster as part of a distributed destination set, as discussed in "Distributed Destination Within a Cluster" on page 4-5.

What About Failover?

This release of WebLogic Server supports migration at the server level—a complete server instance, and all of the services it hosts can be migrated to another machine, either automatically, or manually. For more information, see "Server Migration" in *Using WebLogic Server Clusters*.

In a clustered environment, WebLogic Server also offers service continuity in the event of a single server failure by allowing you to configure distributed destinations, where the members of the unit are usually distributed across multiple servers within a cluster, with each member belonging to a separate JMS server. In addition, implementing the Migratable Service feature, will ensure that exactly-once services, like JMS, do not introduce a single point of failure for dependent applications in the cluster,

BEA also recommends implementing high-availability clustering software such as VERITAS™ Cluster Server, which provides an integrated, out-of-the-box solution for BEA WebLogic Server-based applications. Some other recommended high-availability software solutions include SunCluster, IBM HACMP, or the equivalent.

For information about performing a manual failover, refer to "Recovering From a WebLogic Server Failure" in *Programming WebLogic JMS*.

Configuring Migratable Targets for JMS Servers

Note: This release of WebLogic Server also supports migration at the server level—a complete server instance, and all of the services it hosts can be migrated to another machine, either automatically, or manually. For more information, see "Server Migration" in *Using WebLogic Server Clusters*.

As singleton service, JMS is not active on all server instances in a cluster. It is instead pinned to a single server in the cluster to preserve data consistency. To ensure that singleton services, such as JMS and the JTA transaction recovery service, do not introduce a single point of failure for dependent applications in the cluster, WebLogic Server can be configured to migrate singleton services to any server instance 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 manual migrations in response to a WebLogic Server failure within the cluster.

For more information about the migration of singleton services, see "Service Migration" in *Using WebLogic Server Clusters*.

Configuration Steps for JMS Service Migration

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

1. Optionally, familiarize yourself with how server migration for pinned services works by reading "Service Migration" in *Using WebLogic Server Clusters*.

2. For JMS 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 the store. For more information about migrating persistent stores, see "Persistent Store High Availability" on page 4-8.

3. Configure a migratable target server for the cluster that can potentially host a JMS server, as described in "Configure Migratable Targets for Pinned Services" in *Using WebLogic Server Clusters*.

   Note: You must set a unique Listen Address value for the migratable target server instance that will host a migrated the JMS server; otherwise, the migration will fail.

4. Identify a migratable target server instance on which to deploy a JMS server as described in "Deploying JMS to a Migratable Target Server Instance" in *Using WebLogic Server Clusters*.

   Note: When a migratable target server boots, the JMS server automatically boots as well on the user-preferred server in the cluster.

5. You can also manually migrate a JMS server and all of its destinations before performing server maintenance or to a healthy server if the host server fails. For more information, see "Migrating a Pinned Service To a Target Service Instance" in *Using WebLogic Server Clusters*.

   Note: A JMS server's distributed destination members can migrate to another server instance within a cluster—even when the target server instance is already hosting a JMS server with its own distributed destination members. For more information about distributed destination failover, see "Distributed Destination Failover" on page 4-25.

Persistent Store High Availability

As discussed in "What About Failover?" on page 4-6, a JMS server can be migrated as part of the "server-level" migration feature, or, as discussed in "Configuration Steps for JMS Service Migration" on page 4-7, as part of a "service-level" migration for migratable services like JMS and the JTA transaction recovery service. However, for continued data integrity, file-based persistent stores (default or custom) must be configured on a shared disk that is available to the migratable target servers in the cluster.

For more information on high availability for persistent stores, see *Using the WebLogic Persistent Store* in *Configuring WebLogic Server Environments*.

Using the WebLogic Path Service

The WebLogic Server Path Service is a persistent map that can be used to store the mapping of a group of messages in a Message Unit-of-Order to a messaging resource by pinning messages to a distributed queue member or a store-and-forward path. For more information on the Message Unit-of-Order feature, see "Using Message Unit-of-Order" in *Programming WebLogic JMS*

To configure a path service, see Configure path services in *Administration Console Online Help.*

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

- Each path service mapping is stored in a persistent store. When configuring a path service, select a persistent store that takes advantage of a high-availability solution. See "Persistent Store High Availability" on page 4-8.
- If one or more producers send messages using the same Unit-of-Order name, all messages they produce will share the same path entry and have the same member queue destination.
- If the required route for a Unit-of-Order name is unreachable, the producer sending the message will throw a `JMSOrderException`. The exception is thrown because the JMS messaging system can not meet the quality-of-service required — only one distributed destination member consumes messages for a particular Unit-of-Order.
- A path entry is automatically deleted when the last producer and last message reference are deleted.
- Depending on your system, using the Path Service may slow system throughput due to a remote disk operations to create, read, and delete path entries.
- A distributed queue and its individual members each represent a unique destination. For example:

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

  - If message M1 is sent to DXQ1, it uses the Path Service to define a route to Q2.
  - If message M1 is sent directly to Q2, no routing by the Path Service is performed. This is because the application selected Q2 directly and the system was not asked to pick a member from a distributed destination.
  - If you want the system to use the Path Service, send messages to the distributed destination. If not, send directly to the member.

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

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

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

Accessing Foreign Server Providers

WebLogic JMS enables you to reference foreign (that is, third-party) JMS providers within a local WebLogic Server JNDI tree. With a Foreign Server resource, you can quickly map a foreign JMS provider so that its associated connection factories and destinations appear in the WebLogic JNDI tree as local JMS objects. A Foreign Server resource can also be used to reference remote instances of WebLogic Server in another cluster or domain in the local WebLogic JNDI tree.

Note: In order to use the Foreign Providers feature to reference remote WebLogic Server clusters or domains, you must have a clustered JMS license, which allows a connection factory and a destination to be on different server instances. If you do not have a valid clustered JMS license, contact your BEA sales representative.

For more information on integrating remote and foreign JMS providers, see "Enhanced 2EE Support for Using WebLogic JMS With EJBs and Servlets" in *Programming WebLogic JMS*.

These sections provide more information on how a Foreign Server works and a sample configuration for accessing a remote MQSeries JNDI provider.

- "How WebLogic JMS Accesses Foreign JMS Providers" on page 4-11
- "Creating Foreign Servers" on page 4-12
- "Creating Foreign Connection Factories" on page 4-12
- "Creating a Foreign Destination" on page 4-13
- "Sample Configuration for MQSeries JNDI" on page 4-13

How WebLogic JMS Accesses Foreign JMS Providers

When a foreign JMS server is deployed, it creates local connection factory and destination objects in WebLogic Server JNDI. Then when a foreign connection factory or destination object is looked up on the local server, that object performs the actual lookup on the remote JNDI directory, and the foreign object is returned from that directory.

This method makes it easier to configure multiple WebLogic Messaging Bridge destinations, since the foreign server moves the JNDI Initial Context Factory and Connection URL configuration details outside of your Messaging Bridge destination configurations. You need only provide the foreign Connection Factory and Destination JNDI name for each object.

For more information on configuring a Messaging Bridge, see *Configuring and Managing WebLogic Messaging Bridge*.

The ease-of-configuration concept also applies to configuring WebLogic Servlets, EJBs, and Message-Driven Beans (MDBs) with WebLogic JMS. For example, the `weblogic-ejb-jar.xml` file in the MDB can have a local JNDI name, and you can use the foreign JMS server to control where the MDB receives messages from. For example, you can deploy the MDB in one environment to talk to one JMS destination and server, and you can deploy the same `weblogic-ejb-jar.xml` file to a different server and have it talk to a different JMS destination without having to unpack and edit the `weblogic-ejb-jar.xml` file.

Creating Foreign Servers

A *Foreign Server* represents a JNDI provider that is outside the WebLogic JMS server. It contains information that allows a local WebLogic Server instance to reach a remote JNDI provider, thereby allowing for a number of foreign connection factory and destination objects to be defined on one JNDI directory.

The WebLogic Server Administration Console enables you to configure, modify, target, and delete foreign server resources in a system module. For a road map of the foreign server tasks, see “Configure foreign servers” in the *Administration Console Online Help*.

Some foreign server options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected. For more information about the default values for all foreign server options, see `ForeignServerBean` in the *WebLogic Server MBean Reference*.

After defining a foreign server, you can configure connection factory and destination objects. You can configure one or more connection factories and destinations (queues or topics) for each foreign server.

Creating Foreign Connection Factories

A *Foreign Connection Factory* contains the JNDI name of the connection factory in the remote JNDI provider, the JNDI name that the connection factory is mapped to in the local WebLogic Server JNDI tree, and an optional user name and password.

The foreign connection factory creates non-replicated JNDI objects on each WebLogic Server instance that the parent foreign server is targeted to. (To create the JNDI object on every node in a cluster, target the foreign server to the cluster.)

Creating a Foreign Destination

A *Foreign Destination* represents either a queue or a topic. It contains the destination JNDI name that is looked up on the foreign JNDI provider and the JNDI name that the destination is mapped to on the local WebLogic Server. When the foreign destination is looked up on the local server, a lookup is performed on the remote JNDI directory, and the destination object is returned from that directory.

Sample Configuration for MQSeries JNDI

The following table provides a possible a sample configuration when accessing a remote MQSeries JNDI provider.

Table 4-1 Sample MQSeries Configuration

Foreign JMS Object	Option Names	Sample Configuration Data
Foreign Server	Name	MQJNDI
	JNDI Initial Context Factory	com.sun.jndi.fscontext.RefFSContextFactory
	JNDI Connection URL	file:/MQJNDI/
	JNDI Properties	(If necessary, enter a comma-separated name=value list of properties.)
Foreign Connection Factory	Name	MQ_QCF
	Local JNDI Name	mqseries.QCF
	Remote JNDI Name	QCF
	Username	weblogic_jms
	Password	weblogic_jms
Foreign Destination 1	Name	MQ_QUEUE1
	Local JNDI Name	mqseries.QUEUE1
	Remote JNDI Name	QUEUE_1
Foreign Destination 2	Name	MQ_QUEUE2
	Local JNDI Name	mqseries.QUEUE2
	Remote JNDI Name	QUEUE_2

Configuring Distributed Destinations

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

Note: In order to use the Distributed Destinations feature across multiple WebLogic Server instances, you must have a clustered JMS license, which allows the targeting of destination members to multiple server instances. If you do not have a valid clustered JMS license, contact your BEA sales representative.

These sections provide information on how to create, monitor, and load balance distributed destinations:

- "Uniform Distributed Destinations vs. Weighted Distributed Destinations" on page 4-14
- "Creating Uniform Distributed Destinations" on page 4-15
- "Creating Weighted Distributed Destinations" on page 4-17
- "Monitoring UDD Members" on page 4-17
- "Load Balancing Messages Across a Distributed Destination" on page 4-18
- "Distributed Destination Migration" on page 4-24
- "Distributed Destination Failover" on page 4-25

Uniform Distributed Destinations vs. Weighted Distributed Destinations

This release introduces a new type of distributed destination, termed *uniform distributed destination* (UDD), that greatly simplifies the management and development of distributed destination applications. In prior releases, in order to create a distributed destination, an administrator often needed to manually configure physical destinations to function as members of a distributed destination. This method provided the flexibility to create members that were intended to carry extra message load or have extra capacity; however, such differences often led to administrative and application problems because such a weighted distributed destination was not deployed consistently across a cluster. This type of distributed destination is now officially referred to as a *weighted distributed destination* (or WDD).

Using uniform distributed destinations, you no longer need to create or designate destination members, but instead rely on WebLogic Server to uniformly create the necessary members on the JMS servers to which a JMS module is targeted. This feature ensures the consistent configuration of all distributed destination parameters, particularly in regards to weighting, security, persistence, paging, and quotas.

The weighted distributed destination feature is still available for users who prefer to manually fine-tune distributed destination members. However, BEA strongly recommends configuring uniform distributed destinations to avoid possible administrative and application problems due to a weighted distributed destinations not being deployed consistently across a cluster.

For more information about using a distributed destination with your applications, see "Using Distributed Destinations" in *Programming WebLogic JMS*.

Creating Uniform Distributed Destinations

The WebLogic Server Administration Console enables you to configure, modify, target, and delete UDD resources in a system module. By leaving the "Allocate Members Uniformly" check box selected, the WebLogic Server automatically creates uniformly-configured destination members on selected JMS servers, or on all JMS servers on a target server or cluster.

For a road map of the uniform distributed destination tasks, see the following topics in the *Administration Console Online Help*:

- Configure uniform distributed queues
- Configure uniform distributed topics

Some uniform distributed destination options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected. For more information about the default values for all uniform distributed destination options, see the following entries in the *WebLogic Server MBean Reference*:

- `UniformDistributedQueueBean`
- `UniformDistributedTopicBean`

Note: Because error destinations must be targeted to the same JMS server as the destination(s) it is associated with, error destinations cannot be used with distributed destinations since distributed destination members are targeted to multiple JMS servers.

Targeting Uniform Distributed Queues and Topics

Unlike standalone queues and topics, which can only be deployed to a specific JMS server in a domain, UDDs can be targeted to one or more JMS servers, one or more WebLogic Server instances, or to a cluster, since the purpose of UDDs is to distribute its members on every JMS server in a domain. For example, targeting a UDD to a cluster ensures that a member is uniformly configured on every JMS server in the cluster.

Caution: Changing the targets of a UDD can lead to the removal of a member destination and the unintentional loss of messages.

You can also use subdeployment groups when configuring UDDs to link specific resources with the distributed members. For example, if a system module named *jmssysmod-jms.xml*, is targeted to three WebLogic Server instances: *wlserver1*, *wlserver2*, and *wlserver3*, each with a configured JMS server, and you want to target a uniform distributed queue and a connection factory to each server instance, you can group the UDQ and connection factory in a subdeployment named *servergroup*, to ensure that these resources are always linked to the same server instances.

Here's how the *servergroup* subdeployment resources would look in *jmssysmod-jms.xml*:

```
<weblogic-jms xmlns="http://www.bea.com/ns/weblogic/90">
  <connection-factory name="connfactory">
    <sub-deployment-name>servergroup</sub-deployment-name>
    <jndi-name>jms.connectionfactory.CF</jndi-name>
  </connection-factory>
 <uniform-distributed-queue name="UniformDistributedQueue">
    <sub-deployment-name>servergroup</sub-deployment-name>
    <jndi-name>jms.queue.UDQ</jndi-name>
    <forward-delay>10</forward-delay>
 </uniform-distributed-queue>
</weblogic-jms>
```

And here's how the *servergroup* subdeployment targeting would look in the domain's configuration file:

```
  <jms-system-resource>
   <name>jmssysmod-jms</name>
   <target>cluster1,</target>
   <sub-deployment>
     <name>servergroup</name>
     <target>wlserver1,wlserver2,wlserver3</target>
   </sub-deployment>
```

```
    <descriptor-file-name>jms/jmssysmod-jms.xml</descriptor-file-name>
  </jms-system-resource>
```

Pausing and Resuming Message Operations on UDD Members

You can pause and resume message production, insertion, and/or consumption operations on a uniform distributed destinations, either programmatically (using JMX and the runtime MBean API) or administratively (using the Administration Console). In this way, you can control the JMS subsystem behavior in the event of an external resource failure that would otherwise cause the JMS subsystem to overload the system by continuously accepting and delivering (and redelivering) messages.

For more information on the "pause and resume" feature, see "Controlling Message Operations on Destinations" on page 8-15.

Monitoring UDD Members

Runtime statistics for uniform distributed destination members can be monitored via the Administration console, as described in "Monitoring JMS Statistics" on page 7-3.

Creating Weighted Distributed Destinations

The WebLogic Server Administration Console enables you to configure, modify, target, and delete WDD resources in a system module. When configuring a distributed topic or distributed queue, clearing the "Allocate Members Uniformly" check box allows you to manually select existing queues and topics to add to the distributed destination, and to fine-tune the weighting of resulting distributed destination members.

For a road map of the weighted distributed destination tasks, see the following topics in the *Administration Console Online Help*:

- Create weighted distributed queues
- Create weighted distributed topics

Some weighted distributed destination options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected. For more information about the default values for all weighted distributed destination options, see the following entries in the *WebLogic Server MBean Reference*:

- `DistributedQueueBean`
- `DistributedTopicBean`

Unlike UDDs, WDD members cannot be monitored with the Administration Console or though runtime MBeans. Also, WDDs members cannot be uniformly targeted to JMS server or WebLogic Server instances in a domain. Instead, new WDD members must be manually configured on such instances, and then manually added to the WDD.

Load Balancing Messages Across a Distributed Destination

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

Load Balancing Options

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

Round-Robin Distribution

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.

For weighted distributed destinations only, 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.

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 4-18. 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 4-18.

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 4-18, WebLogic JMS uses the following heuristics when choosing an instance of a destination.

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

The Server Affinity Enabled parameter on connection factories defines whether a WebLogic Server that is load balancing consumers or producers across multiple member destinations in a distributed destination set, will first attempt to load balance across any other local destination members that are also running on the same WebLogic Server.

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

To disable server affinity on a connection factory:

1. Follow the directions for navigating to the JMS Connection Factory → Configuration → General page in "Configure load balancing parameters" in the *Administration Console Online Help*.
2. Define the Server Affinity Enabled field as follows:
   - If the Server Affinity Enabled check box is selected (True), then a WebLogic Server that is load balancing consumers or producers across multiple physical destinations in a distributed destination set, will first attempt to load balance across any other physical destinations that are also running on the same WebLogic Server.
   - If the Server Affinity Enabled check box is not selected (False), then a WebLogic Server will load balance consumers or producers across physical destinations in a distributed destination set and disregard any other physical destinations also running on the same WebLogic Server.
3. Click Save.

For more information about how the Server Affinity Enabled setting affects the load balancing among the members of a distributed destination, see "How Distributed Destination Load Balancing Is Affected When Server Affinity Is Enabled" on page 4-22.

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.

Paused Distributed Destination Members

When distributed destinations are paused for message production or insertion, they are not considered for message production. Similarly, when destinations are paused for consumption, they are not considered for message production.

For more information on pausing message operations on destinations, see "Controlling Message Operations on Destinations" on page 8-15.

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.

For instructions on how to directly access uniform and weighted distributed destination members, see "Accessing Distributed Destination Members" in *Programming WebLogic JMS*.

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 parameter disabled. To ensure a fair distribution of the messaging load among a distributed destination, the initial physical destination (queue or topic) used by producers is always chosen at random from among the distributed destination members.

To disable load balancing on a connection factory:

1. Follow the directions for navigating to the JMS Connection Factory → Configuration → General page in "Configure load balancing parameters" in the *Administration Console Online Help*.
2. Define the setting of the Load Balancing Enabled field using the following guidelines:
   - `Load Balancing Enabled = True`
     For `Queue.sender.send()` methods, non-anonymous producers are load balanced on *every* invocation across the distributed queue members.

     For `TopicPublish.publish()` methods, non-anonymous producers are always pinned to the same physical topic for every invocation, irrespective of the Load Balancing Enabled setting.

- `Load Balancing Enabled = False`
  Producers always produce to the same physical destination until they fail. At that point, a new physical destination is chosen.

3. Click Save.

Note: Depending on your implementation, the setting of the Server Affinity Enabled attribute can affect load balancing preferences for distributed destinations. For more information, see "How Distributed Destination Load Balancing Is Affected When Server Affinity Is Enabled" on page 4-22.

Anonymous producers (producers that do not designate a destination when created), are load-balanced each time they switch destinations. If they continue to use the same destination, then the rules for non-anonymous producers apply (as stated previously).

How Distributed Destination Load Balancing Is Affected When Server Affinity Is Enabled

Table 4-2 explains how the setting of a connection factory's Server Affinity Enabled parameter affects the load balancing preferences for distributed destination members. The order of preference depends on the type of operation and whether or not durable subscriptions or persistent messages are involved.

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

For more information, refer to the "Load Balancing for EJBs and RMI Objects" and "Initial Context Affinity and Server Affinity for Client Connections" sections in *Using WebLogic Server Clusters.*

Table 4-2 Server Affinity Load Balancing Preferences

When the operation is...	And Server Affinity Enabled is...	Then load balancing preference is given to a...
• `createReceiver()` for queues • `createSubscriber()` for topics	True	1. local member without a consumer 2. local member 3. remote member without a consumer 4. remote member
`createReceiver()` for queues	False	1. member without a consumer 2. member
`createSubscriber()` for topics (**Note:** non-durable subscribers)	True or False	1. local member without a consumer 2. local member
• `createSender()` for queues • `createPublisher()` for topics	True or False	There is no separate machinery for load balancing a JMS producer creation. JMS producers are created on the server on which your JMS connection is load balanced or pinned. For more information about load balancing JMS connections created via a connection factory, refer to the "Load Balancing for EJBs and RMI Objects" and "Initial Context Affinity and Server Affinity for Client Connections" sections in *Using WebLogic Server Clusters*.
For persistent messages using `QueueSender.send()`	True	1. local member with a consumer and a store 2. remote member with a consumer and a store 3. local member with a store 4. remote member with a store 5. local member with a consumer 6. remote member with a consumer 7. local member 8. remote member

Table 4-2 Server Affinity Load Balancing Preferences

When the operation is…	And Server Affinity Enabled is…	Then load balancing preference is given to a…
For persistent messages using `QueueSender.send()`	False	1. member with a consumer and a store 2. member with a store 3. member with a consumer 4. member
For non-persistent messages using `QueueSender.send()`	True	1. local member with a consumer 2. remote member with a consumer 3. local member 4. remote member
For non-persistent messages: • `QueueSender.send()` • `TopicPublish.publish()`	False	1. member with a consumer 2. member
`createConnectionConsumer()` for session pool queues and topics	True or False	1. local member *only* **Note:** Session pools are now used rarely, as they are not a required part of the J2EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are simpler, easier to manage, and more capable.

Distributed Destination Migration

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

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

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

For more information about the configuring JMS migratable targets, see "Configuring Migratable Targets for JMS Servers" on page 4-7.

Distributed Destination Failover

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

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

For more information about procedures for recovering from a WebLogic Server failure, see "Recovering From a WebLogic Server Failure" in *Programming WebLogic JMS*.

Configuring JMS Application Modules for Deployment

JMS resources can be configured and managed as deployable application modules, similar to standard J2EE modules. Deployed JMS application modules are owned by the developer who created and packaged the module, rather than the administrator who deploys the module; therefore, the administrator has more limited control over deployed resources.

For example, administrators can only modify (override) certain properties of the resources specified in the module using the deployment plan (JSR-88) at the time of deployment, but they cannot dynamically add or delete resources. As with other J2EE modules, configuration changes for an application module are stored in a deployment plan for the module, leaving the original module untouched.

These sections explain how to configure JMS application modules for deployment, including globally available standalone modules and modules packaged with a J2EE application.

JMS Schema

In support of the new modular deployment model for JMS resources in WebLogic Server 9.0, BEA now provides a schema for defining WebLogic JMS resources: `weblogic-jmsmd.xsd`. When you create JMS modules (descriptors), the modules must conform to this schema. IDEs and other tools can validate JMS modules based on the schema.

The `weblogic-jmsmd.xsd` schema is available online at `http://www.bea.com/ns/weblogic/90/weblogic-jmsmd.xsd`.

For an explanation of the JMS resource definitions in the schema, see the corresponding system module beans in the "System Module MBeans" folder of the *WebLogic Server MBean Reference*. The root bean in the JMS module that represents an entire JMS module is named JMSBean.

Deploying JMS Modules That Are Packaged In an Enterprise Application

JMS application modules can be packaged as part of an Enterprise Application, as a *packaged* resource. Packaged modules are bundled with an EAR or exploded EAR directory, and are referenced in the `weblogic-application.xml` descriptor.

The packaged JMS module is deployed along with the Enterprise Application, and the resources defined in this module can optionally be made available only to the enclosing application (i.e., as an *application-scoped* resource). Such modules are particularly useful when packaged with EJBs (especially MDBs) or Web Applications that use JMS resources. Using packaged modules ensures that an application always has required resources and simplifies the process of moving the application into new environments.

Creating Packaged JMS Modules

You create packaged JMS modules using an enterprise-level IDE or another development tool that supports editing of XML descriptor files. You then deploy and manage standalone modules using JSR 88-based tools, such as the `weblogic.Deployer` utility or the WebLogic Administration Console**.**

Note: You can create a packaged JMS module using the Administration Console, then copy the resulting `XML` file to another directory and rename it, using "`-jms.xml`" as the file suffix.

JMS Packaged Module Requirements

Inside the EAR file, a JMS module must meet the following criteria:

- Conforms to the `weblogic-jmsmd.xsd` schema
- Uses "`-jms.xml`" as the file suffix (for example, *`MyJMSDescriptor`*`-jms.xml`)
- Uses a name that is unique within the WebLogic domain and a path that is relative to the root of the J2EE application

Main Steps for Creating Packaged JMS Modules

Follow these steps to configure a packaged JMS module:

1. If necessary, create a JMS server to target the JMS module to, as explained in "Configure JMS Servers" in the *Administration Console Online Help*.
2. Create a JMS system module and configure the necessary resources, such as queues or topics, as described in "Configure JMS system modules and add JMS resources" in the *Administration Console Online Help*.
3. The system module is saved in `config\jms` subdirectory of the domain directory, with a "`-jms.xml`" suffix.
4. Copy the system module to a new location, and then:
   a. Give the module a unique name within the domain namespace.
   b. Delete the `JNDI-Name` attribute to make the module *application-scoped* to only the application.
5. Add references to the JMS resources in the module to all applicable J2EE application component's descriptor files, as described in "Referencing a Packaged JMS Module In Deployment Descriptor Files" on page 5-3.
6. Package all application modules in an EAR, as described in "Packaging an Enterprise Application With a JMS Module" on page 5-8.
7. Deploy the EAR, as described in "Deploying a Packaged JMS Module" on page 5-8.

Referencing a Packaged JMS Module In Deployment Descriptor Files

When you package a JMS module with an enterprise application, you must reference the JMS resources within the module in all applicable descriptor files of the J2EE application components, including:

- The WebLogic enterprise descriptor file, `weblogic-application.xml`
- Any WebLogic deployment descriptor file, such as `weblogic-ejb-jar.xml` or `weblogic.xml`
- Any J2EE descriptor file, such as EJB (`ejb-jar.xml`) or WebApp (`web.xml`) files

Referencing JMS Modules In a weblogic-application.xml Descriptor

When including JMS modules in an enterprise application, you must list each JMS module as a module element of type JMS in the `weblogic-application.xml` descriptor file packaged with the application, and a path that is relative to the root of the J2EE application. Here's an example of a reference to a JMS module name *Workflows*:

```
<module>
  <name>Workflows</name>
  <type>JMS</type>
  <path>jms/Workflows-jms.xml</path>
</module>
```

Referencing JMS Resources In a WebLogic Application

Within any `weblogic-`*`foo`* descriptor file, such as EJB (`weblogic-ejb-jar.xml`) or WebApp (`weblogic.xml`), the name of the JMS module is followed by a pound (#) separator character, which is followed by the name of the resource inside the module. For example, a JMS module named *Workflows* containing a queue named *OrderQueue*, would have a name of *Workflows#OrderQueue*.

```
<resource-env-description>
  <resource-env-ref-name>jms/OrderQueue</resource-env-ref-name>
  <resource-link>Workflows#OrderQueue</resource-link>
</resource-env-description>
```

Note that the `<resource-link>` element is unique to WebLogic Server, and is how the resources that are defined in a JMS Module are referenced (linked) from the various other J2EE Application components.

Referencing JMS Resources In a J2EE Application

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

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

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

```
<resource-ref>
  <res-ref-name>jms/OrderQueueFactory</res-ref-name>
  <res-type>javax.jms.ConnectionFactory</res-type>
</resource-ref>
<resource-env-ref>
  <res-env-ref-name>jms/OrderQueue</res-env-ref-name>
  <res-env-ref-type>javax.jms.Queue</res-env-ref-type>
 </resource-env-ref>
```

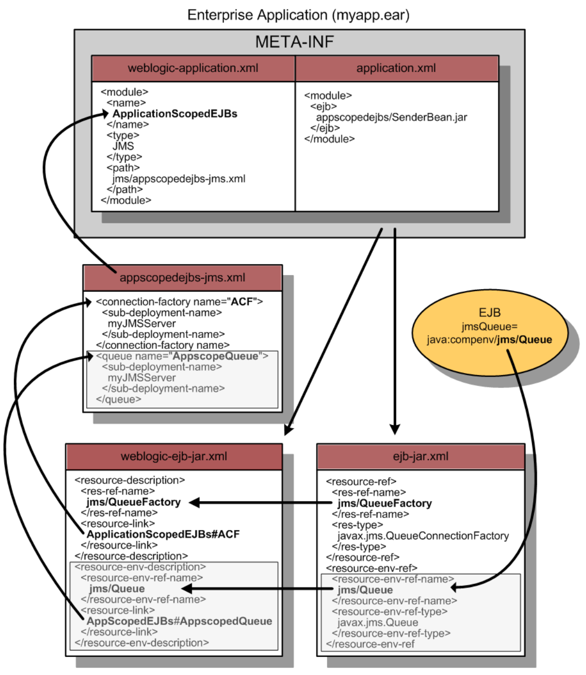
Sample of a Packaged JMS Module In an EJB Application

The following code snippet is an example of the packaged JMS module, `appscopedejbs-jms.xml`, referenced by the descriptor files in Figure 5-1 below.

```
<weblogic-jms xmlns="http://www.bea.com/ns/weblogic/90">
  <connection-factory name="ACF">
  </connection-factory>
  <queue name="AppscopeQueue">
  </queue>
</weblogic-jms>
```

Figure 5-1 illustrates how a JMS connection factory and queue resources in a packaged JMS module are referenced in an EJB EAR file.

Figure 5-1 Relationship Between a JMS Module and Descriptors In an EJB Application



```
<module>
  <name>
   ApplicationScopedEJBs
  </name>
  <type>
   JMS
  </type>
  <path>
   jms/appscopedejbs-jms.xml
  </path>
</module>
```

```
<module>
  <ejb>
   appscopedejbs/SenderBean.jar
  </ejb>
</module>
```

```
<connection-factory name="ACF">
  <sub-deployment-name>
   myJMSServer
  </sub-deployment-name>
</connection-factory name>
<queue name="AppscopeQueue">
  <sub-deployment-name>
   myJMSServer
  </sub-deployment-name>
</queue>
```

```
<resource-description>
  <res-ref-name>
   jms/QueueFactory
  </res-ref-name>
  <resource-link>
   ApplicationScopedEJBs#ACF
  </resource-link>
</resource-description>
<resource-env-description>
  <resource-env-ref-name>
   jms/Queue
  </resource-env-ref-name>
  <resource-link>
   AppScopedEJBs#AppscopedQueue
  </resource-link>
</resource-env-description>
```

```
<resource-ref>
  <res-ref-name>
   jms/QueueFactory
  </res-ref-name>
  <res-type>
   javax.jms.QueueConnectionFactory
  </res-type>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>
   jms/Queue
  </resource-env-ref-name>
  <resource-env-ref-type>
   javax.jms.Queue
  </resource-env-ref-type>
</resource-env-ref
```

Packaged JMS Module References In weblogic-application.xml

When including JMS modules in an enterprise application, you must list each JMS module as a module element of type JMS in the `weblogic-application.xml` descriptor file packaged with the application, and a path that is relative to the root of the application. For example:

```
<module>
  <name>AppScopedEJBs</name>
  <type>JMS</type>
  <path>jms/appscopedejbs-jms.xml</path>
</module>
```

Packaged JMS Module References In ejb-jar.xml

If EJBs in your application use connection factories through a JMS module packaged with the application, you must list the JMS module as a `res-ref` element and include the `res-ref-name` and `res-type` parameters in the `ejb-jar.xml` descriptor file packaged with the EJB. This way, the EJB can lookup the JMS Connection Factory in the application's local context. For example:

```
<resource-ref>
  <res-ref-name>jms/QueueFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
 </resource-ref>
```

The `res-ref-name` element maps the resource name (used by `java:comp/env`) to a module referenced by an EJB. The `res-type` element specifies the module type, which in this case, is `javax.jms.QueueConnectionFactory`.

If EJBs in your application use Queues or Topics through a JMS module packaged with the application, you must list the JMS module as a `resource-env-ref` element and include the `resource-env-ref-name` and `resource-env-ref-type` parameters in the `ejb-jar.xml` descriptor file packaged with the EJB. This way, the EJB can lookup the JMS Queue or Topic in the application's the local context. For example:

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

The `resource-env-ref-name` element maps the destination name to a module referenced by an EJB. The `res-type` element specifies the name of the Queue, which in this case, is `javax.jms.Queue`.

Packaged JMS Module References In weblogic-ejb-jar.xml

You must list the referenced JMS module as a `res-ref-name` element and include the `resource-link` parameter in the `weblogic-ejb-jar.xml` descriptor file packaged with the EJB.

```
<resource-description>
  <res-ref-name>jms/QueueFactory</res-ref-name>
  <resource-link>AppScopedEJBs#ACF</resource-link>
 </resource-description>
```

The `res-ref-name` element maps the connection factory name to a module referenced by an EJB. In the `resource-link` element, the JMS module name is followed by a pound (#) separator character, which is followed by the name of the resource inside the module. So for this example, the JMS module *AppScopedEJBs* containing the connection factory *ACF*, would have a name *AppScopedEJBs#ACF*.

Continuing the example above, the `res-ref-name` element also maps the Queue name to a module referenced by an EJB. And in the `resource-link` element, the queue *AppScopedQueue*, would have a name *AppScopedEJBs#AppScopedQueue*, as follows:

```
<resource-env-description>
  <resource-env-ref-name>jms/Queue</resource-env-ref-name>
  <resource-link>AppScopedEJBs#AppScopedQueue</resource-link>
 </resource-env-description>
```

Packaging an Enterprise Application With a JMS Module

You package an application with a JDBC module as you would any other enterprise application. See "Packaging Applications Using wlpackage" in *Developing Applications with WebLogic Server*.

Deploying a Packaged JMS Module

The deployment of packaged JMS modules follows the same model as all other components of an application: individual modules can be deployed to a single server, a cluster, or individual members of a cluster.

A recommended best practice for other application components is to use the `java:comp/env` JNDI environment in order to retrieve references to JMS entities, as described in "Referencing JMS Resources In a J2EE Application" on page 5-5. (However, this practice is not required.)

By definition, packaged JMS modules are included in an enterprise application, and therefore are deployed when you deploy the enterprise application. For more information about deploying applications with packaged JMS modules, see "Deploying Applications Using wldeploy" in *Developing Applications with WebLogic Server*.

Deploying Standalone JMS Modules

A JMS application module can be deployed as a standalone resource using the `weblogic.Deployer` utility or the Administration Console, in which case the module is typically available to the server or cluster targeted during the deployment process. JMS modules deployed in this manner are called *standalone modules*. Depending on how they are targeted, the resources inside standalone JMS modules are globally available in a cluster or locally on a server instance.

Standalone JMS modules promote sharing and portability of JMS resources. You can create a JMS module and distribute it to other developers. Standalone JMS modules can also be used to move JMS information between domains, such as between the development domain and the production domain, without extensive manual JMS reconfiguration.

Creating Standalone JMS Modules

You can create JMS standalone modules using an enterprise-level IDE or another development tool that supports editing XML descriptor files. You then deploy and manage standalone modules using WebLogic Server tools, such as the `weblogic.Deployer` utility or the WebLogic Administration Console.

Note: You can create a JMS application module using the Administration Console, then copy the module as a template for use in your applications, using "`-jms.xml`" as the file suffix. You must also change the `Name` and `JNDI-Name` elements of the module before deploying it with your application to avoid a naming conflict in the namespace.

JMS Standalone Module Requirements

A standalone JMS module must meet the following criteria:

- Conforms to the `weblogic-jmsmd.xsd` schema
- Uses "`-jms.xml`" as the file suffix (for example, *`MyJMSDescriptor`*`-jms.xml`)
- Uses a name that is unique within the WebLogic domain (cannot conflict with JMS system modules)

Main Steps for Creating Standalone JMS Modules

Follow these steps to configure a standalone JMS module:

1. If necessary, create a JMS server to target the JMS module to, as explained in "Configure JMS Servers" in the *Administration Console Online Help*.
2. Create a JMS system module and configure the necessary resources, such as queues or topics, as described in "Configure JMS system modules and add JMS resources" in the *Administration Console Online Help*.
3. The system module is saved in `config\jms` subdirectory of the domain directory, with a "`-jms.xml`" suffix.
4. Copy the system module to a new location and then:
   a. Give the module a unique name within the domain namespace.
   b. To make the module *globally available*, uniquely rename the `JNDI-Name` attributes of the resources in the module.
   c. If necessary, modify any other tunable values, such as destination thresholds or connection factory flow control parameters.
5. Deploy the module, as described in "Deploying Standalone JMS Modules" on page 5-11.

Sample of a Simple Standalone JMS Module

The following code snippet is an example of simple standalone JMS module.

```
<weblogic-jms xmlns="http://www.bea.com/ns/weblogic/90">
  <connection-factory name="exampleStandAloneCF">
    <jndi-name>exampleStandAloneCF</jndi-name>
  </connection-factory>
 <queue name="ExampleStandAloneQueue">
    <jndi-name>exampleStandAloneQueue</jndi-name>
  </queue>
</weblogic-jms>
```

Deploying Standalone JMS Modules

The command-line for using the `weblogic.Deployer` utility to deploy a standalone JMS module (using the example above) would be:

```
java weblogic.Deployer -adminurl http://localhost:7001 -user weblogic
-password weblogic \
-name ExampleStandAloneJMS \
-targets examplesServer \
-submoduletargets
ExampleStandaloneQueue@examplesJMSServer,ExampleStandaloneCF@examplesServer \
-deploy ExampleStandAloneJMSModule-jms.xml
```

For information about deploying standalone JMS modules, see "Deploying JDBC and JMS Application Modules."

When you deploy a standalone JMS module, an `app-deployment` entry is added to the `config.xml` file for the domain. For example:

```
<app-deployment>
  <name>standalone-examples-jms</name>
  <target>MedRecServer</target>
  <module-type>jms</module-type>
  <source-path>C:\modules\standalone-examples-jms.xml</source-path>
  <sub-deployment>
  ...
  </sub-deployment>
  <sub-deployment>
  ...
  </sub-deployment>
</app-deployment>
```

Note that the `source-path` for the module can be an absolute path or it can be a relative path from the *`domain`* directory. This differs from the `descriptor-file-name` path for a system resource module, which is relative to the *`domain`*`\config` directory.

Tuning Standalone JMS Modules

JMS resources deployed within *standalone modules* can be reconfigured using the using the `weblogic.Deployer` utility or the Administration Console, as long as the resources are considered bindable (such as JNDI names), or tunable (such as destination thresholds). However, standalone resources are not available through WebLogic JMX APIs or the WebLogic Scripting Tool (WLST).

However, standalone JMS modules are available using the basic JSR-88 deployment tool provided with WebLogic Server plug-ins (without using WebLogic Server extensions to the API) to configure, deploy, and redeploy J2EE applications and modules to WebLogic Server. For information about WebLogic Server deployment, see "Understanding WebLogic Server Deployment."

Additionally, standalone resources cannot be dynamically added or deleted with any WebLogic Server utility and must be redeployed.

Using WLST to Manage JMS Servers and JMS System Resources

The WebLogic Scripting Tool (WLST) is a command-line scripting interface that you can use to create and manage JMS servers and JMS system resources. See "Using the WebLogic Scripting Tool" and WLST Sample Scripts in the *WebLogic Scripting Tool*.

Understanding System Modules and Subdeployments

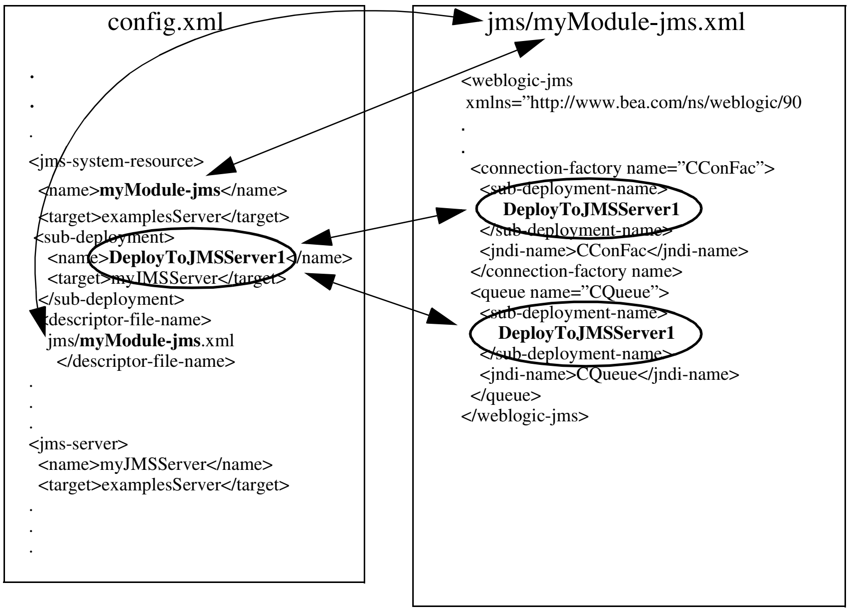
A module is described by the `jms-system-resource` MBean in the `config.xml` file. Basic components of a `jms-system-resource` MBean are:

- `name`—Name of the module.
- `target`—Server, cluster, or migratable target the module is targeted to.
- `sub-deployment`—A mechanism by which JMS module resources (such as queues, topics, and connection factories) are grouped and targeted to a server resource (such as a JMS server instance, WebLogic server instance, or cluster).
- `descriptor-file-name`—Path and filename of the system module file.

The JMS resources of a module are located in a module descriptor file that conforms to the *weblogic-jmsmd.xml* schema. In Figure 6-1, the module is named `myModule-jms.xml` and it contains JMS resource definitions for a connection factory and a queue. The `sub-deployment-name` element is used to group and target JMS resources in the `myModule-jms.xml` file to `targets` in the `config.xml`. You have to provide a value for the `sub-deployment-name` element when using WLST. For more information on subdeployments, see "Targeting JMS Modules and Subdeployment Resources" in *Configuring and Managing WebLogic JMS*. In Figure 6-1, the `sub-deployment-name` *DeployToJMSServer1* is used to group and target the connection factory `CConfac` and the queue `CQueue` in the `myModule-jms` module.

For more information on how to use JMS resources, see "Understanding JMS Resource Configuration" in *Configuring and Managing WebLogic JMS*.

Figure 6-1 Subdeployment Architecture


config.xml

```
.
.
.
<jms-system-resource>
  <name>myModule-jms</name>
  <target>examplesServer</target>
 <sub-deployment>
   <name>DeployToJMSServer1</name>
   <target>myJMSServer</target>
 </sub-deployment>
  <descriptor-file-name>
   jms/myModule-jms.xml
     </descriptor-file-name>
.
.
.
<jms-server>
  <name>myJMSServer</name>
  <target>examplesServer</target>
.
.
.
```

jms/myModule-jms.xml

```
<weblogic-jms
 xmlns="http://www.bea.com/ns/weblogic/90
.
.
  <connection-factory name="CConFac">
    <sub-deployment-name>
      DeployToJMSServer1
    </sub-deployment-name>
    <jndi-name>CConFac</jndi-name>
  </connection-factory name>
  <queue name="CQueue">
    <sub-deployment-name>
      DeployToJMSServer1
    </sub-deployment-name>
    <jndi-name>CQueue</jndi-name>
  </queue>
</weblogic-jms>
```

How to Create JMS Servers and System Resources

Basic tasks you need to perform when creating JMS resources with the WLST are:

- Start an edit session.
- Create a JMS system module that includes JMS system resources, such as queues, topics, and connection factories.
- Create JMS server resources.

After you have established an edit session, use the following steps configure JMS servers and system resources:

1. Get the WebLogic Server MBean object for the server you want to configure resources. For example:

   ```
   servermb=getMBean("Servers/examplesServer")
       if servermb is None:
           print '@@@ No server MBean found'
   ```

2. Create your system resource. For example:

   ```
   jmsMySystemResource = create(myJmsSystemResource,"JMSSystemResource")
   ```

3. Target your system resource to a WebLogic Server instance. For example:

   ```
   jmsMySystemResource.addTarget(servermb)
   ```

4. Get your system resource object. For example:

   ```
   theJMSResource = jmsMySystemResource.getJMSResource()
   ```

5. Create resources for the module, such as queues, topics, and connection factories. For example:

   ```
   connfact1 = theJMSResource.createConnectionFactory(factoryName)
   ```

   ```
   jmsqueue1 = theJMSResource.createQueue(queueName)
   ```

6. Configure resource attributes. For example:

   ```
   connfact1.setJNDIName(factoryName)
   jmsqueue1.setJNDIName(queueName)
   ```

7. Create a subdeployment name for system resources . See "Understanding System Modules and Subdeployments" on page 6-1.For example:

   ```
   connfact1.setSubDeploymentName('DeployToJMSServer1')
   jmsqueue1.setSubDeploymentName('DeployToJMSServer1')
   ```

8. Create a JMS server. For example:

   ```
   jmsserver1mb = create(jmsServerName,'JMSServer')
   ```

9. Target your JMS server to a WebLogic Server instance. For example:

   ```
   jmsserver1mb.addTarget(servermb)
   ```

10. Create a subdeployment object using the value you provided for the `sub-deployment-name` element. This step groups the system resources in module to a `sub-deployment` element in the `config.xml`. For example:

    ```
    subDep1mb = jmsMySystemResource.createSubDeployment('DeployToJMSServer1
    ')
    ```

11. Target the subdeployment to a server resource such as a JMS server instance, WebLogic Server instance, or cluster. For example:

    ```
    subDep1mb.addTarget(jmsserver1mb)
    ```

Listing 6-1 WLST Script to Create JMS Resources

```
"""
This script starts an edit session, creates a JMS Server,
targets the jms server to the server WLST is connected to and creates
a JMS System module with a jms queue and connection factory. The
jms queues and topics are targeted using sub-deployments.
"""

import sys
from java.lang import System

print "@@@ Starting the script ..."

myJmsSystemResource = "CapiQueue-jms"
factoryName = "CConFac"
jmsServerName = "myJMSServer"
queueName = "CQueue"

url = sys.argv[1]
usr = sys.argv[2]
password = sys.argv[3]
```

```
connect(usr,password, url)
edit()
startEdit()

//Step 1
servermb=getMBean("Servers/examplesServer")
    if servermb is None:
        print '@@@ No server MBean found'

else:

    //Step 2
    jmsMySystemResource = create(myJmsSystemResource,"JMSSystemResource")
    //Step 3
    jmsMySystemResource.addTarget(servermb)

    //Step 4
    theJMSResource = jmsMySystemResource.getJMSResource()

    //Step 5
    connfact1 = theJMSResource.createConnectionFactory(factoryName)
    jmsqueue1 = theJMSResource.createQueue(queueName)


    //Step 6
    connfact1.setJNDIName(factoryName)
    jmsqueue1.setJNDIName(queueName)


    //Step 7
    jmsqueue1.setSubDeploymentName('DeployToJMSServer1')
    connfact1.setSubDeploymentName('DeployToJMSServer1')
    //Step 8
    jmsserver1mb = create(jmsServerName,'JMSServer')
    //Step 9
    jmsserver1mb.addTarget(servermb)

    //Step 10
    subDep1mb = jmsMySystemResource.createSubDeployment('DeployToJMSServer
1')
    //Step 11
    subDep1mb.addTarget(jmsserver1mb)
.
.
.
```

How to Modify and Monitor JMS Servers and JMS Resources

You can modify or monitor JMS objects and attributes by using the appropriate method available from the MBean.

- You can modify JMS objects and attributes using the set, target, untarget, and delete methods.
- You can monitor JMS runtime objects using get methods.

For more information, see Navigating and Editing MBeans in the *WebLogic Scripting Tool.*

Listing 6-2 WLST Script to Modify JMS Objects

```
.
.
print '@@@ delete system resource'
jmsMySystemResource = delete("CapiQueue-jms","JMSSystemResource")
print '@@@ delete server'
jmsserver1mb = delete(jmsServerName,'JMSServer')
.
.
.
```

Best Practices when Using WLST to Configure JMS

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

- Trap for Null MBean objects (such as servers, JMS servers, modules) before trying to manipulate the MBean object.
- Use a meaningful name when providing a subdeployment name. For example, the subdeployment name *DeployToJMSServer1* tells you that all subdeployments with this name are deployed to JMSServer1.
- BEA provides sample scripts and utilities to configure WebLogic domain resources using WLST Offline and/or WLST Online. For more information, see the *wlst Project Home* at https://wlst.projects.dev2dev.bea.com/.

Monitoring JMS Statistics and Managing Messages

This release of WebLogic Server includes the WebLogic Diagnostic Service, which is a monitoring and diagnostic service that runs within the WebLogic Server process and participates in the standard server life cycle. This service enables you to create, collect, analyze, archive, and access diagnostic data generated by a running server and the applications deployed within its containers.

For Weblogic JMS, you can use the enhanced runtime statistics to monitor the JMS servers and destination resources in your WebLogic domain to see if there is a problem. If there is a problem, you can use profiling to determine which application is the source of the problem. Once you've narrowed it down to the application, you can then use JMS debugging features to find the problem within the application.

For more information on configuring JMS diagnostic notifications, debugging options, message life cycle logging, and controlling message operations on JMS destinations, see "Troubleshooting WebLogic JMS" on page 8-1.

Message administration tools in this release enhance your ability to view and browse *all* messages, and to manipulate *most* messages in a running JMS Server, using either the Administration Console or through new public runtime APIs. These message management enhancements include message browsing (for sorting), message manipulation (such as create, move, and delete), message import and export, as well as transaction management, durable subscriber management, and JMS client connection management.

The following sections explain how to monitor JMS resource statistics and how to manage your JMS messages from the Administration Console:

- "Monitoring JMS Statistics" on page 7-3

- "Managing JMS Messages" on page 7-6

For more information about the WebLogic Diagnostic Service, see Understanding the WebLogic Diagnostic Service.

Monitoring JMS Statistics

Once WebLogic JMS has been configured, applications can begin sending and receiving messages through the JMS API, as described in "Developing a Basic JMS Application" in *Programming WebLogic JMS*.

You can monitor statistics for the following JMS objects: JMS servers, connections, queue and topic destinations, and JMS server session pools.

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

Monitoring JMS Servers

You can monitor statistics on active JMS servers defined in your domain via the Administration Console or through the JMSServerRuntimeMBean. JMS servers act as management containers for JMS queue and topic resources within JMS modules that are specifically targeted to JMS servers.

For more information on using the Administration Console to monitor JMS servers, see "Monitoring JMS Servers" in the *Administration Console Online Help*.

When monitoring JMS servers with the Administration Console, you can also monitor statistics for active destinations, transactions, connections, and session pools.

Monitoring Active JMS Destinations

You can monitor statistics on all the active destinations currently targeted to a JMS server. JMS destinations identify queue or topic destination types within JMS modules that are specifically targeted to JMS servers.

For more information, see "JMS Servers: Monitoring: Active Destinations" in the *Administration Console Online Help*.

Monitoring Active JMS Transactions

You can monitor view active transactions running on a JMS server.

For more information on the runtime statistics provided for active JMS transactions, see "JMS Servers: Monitoring: Active Transactions" in the *Administration Console Online Help*.

Monitoring Active JMS Connections

You can monitor statistics on all the active JMS connections to a JMS server. A JMS connection is an open communication channel to the messaging system.

For more information on the runtime statistics provided for active JMS server connections, see "JMS Servers: Monitoring: Active Connections" in the *Administration Console Online Help*.

Monitoring Active JMS Session Pools

You can monitor statistics on all the active JMS session pools defined for a JMS server. Session pools enable an application to process messages concurrently.

For more information on the runtime statistics provided for active JMS session pools, see "JMS Servers: Monitoring: Active Session Pools" in the *Administration Console Online Help*.

Monitoring Queues

You can monitor statistics on queue resources in JMS modules via the Administration Console or through the JMSDestinationRuntimeMBean. A JMS queue defines a point-to-point destination type for a JMS server. Queues are used for asynchronous peer communications.A message delivered to a queue will be distributed to one consumer.

For more information on using the Administration Console to monitor queue resources, see "Monitoring Queues in JMS System Modules" in the *Administration Console Online Help*.

You can also use the Administration Console to manage messages on queues, as described in "Managing JMS Messages" on page 7-6.

Monitoring Topics

You can monitor statistics on topic resources in JMS modules via the Administration Console or through the JMSDestinationRuntimeMBean. A JMS topic identifies a publish/subscribe destination type for a JMS server. Topics are used for asynchronous peer communications. A message delivered to a topic will be distributed to all topic consumers.

For more information on using the Administration Console to monitor topic resources, see "Monitor Topics in JMS System Modules" in the *Administration Console Online Help*.

Monitoring Uniform Distributed Queues

You can monitor statistics on uniform distributed queue resources in JMS modules via the Administration Console. A distributed queue resource is a single set of queues that are accessible as a single, logical destination to a client (for example, a distributed topic has its own JNDI name). The members of the unit are usually distributed across multiple servers within a cluster, with each member belonging to a separate JMS server.

For more information on using the Administration Console to monitor uniform distributed queue resources, see "Monitoring Uniform Distributed Queues in JMS System Modules" in the *Administration Console Online Help*.

Monitoring Uniform Distributed Topics

You can monitor statistics on uniform distributed topic resources in JMS modules via the Administration Console. A distributed topic resource is a single set of topics that are accessible as a single, logical destination to a client (for example, a distributed topic has its own JNDI name). The members of the unit are usually distributed across multiple servers within a cluster, with each member belonging to a separate JMS server.

For more information on using the Administration Console to monitor uniform distributed topic resources, see "Monitoring Uniform Distributed Topics in JMS System Modules" in the *Administration Console Online Help*.

Managing JMS Messages

The Weblogic JMS message monitoring and management features allow you to create new messages, delete selected messages, move messages to another queue, export message contents to another file, import message contents from another file, or drain all the messages from the queue.

JMS Message Management via Java APIs

WebLogic Java Management Extensions (JMX) enables you to access the JMSDestinationRuntimeMBean and JMSDurableSubscriberRuntimeMBean to manage messages on JMS queues.

JMS Message Management Using the Administration Console

The JMS Message Management page of the Administration Console summarizes the messages that are available on the queue you are monitoring. You can page through messages and/or retrieve a set of messages that meet filtering criteria you specify. You can also customize the message display to show only the information you need. From this page, you can select a message to display its contents, create new messages, delete one or more messages, move messages, import and export messages, and drain the entire queue. For more information on using the Administration Console to manage messages on a queue, see “Manage queue messages” in the *Administration Console Online Help*.

Each of these functions is described in more detail later in this document.

Monitoring Message Runtime Information

By default, the JMS Message Management page displays the information about each message on a queue in a table with the following columns.

- ID - A unique identifier for the message.
- Type - The JMS message type, such as BytesMessage, TextMessage, StreamMessage, ObjectMessage, MapMessage, or XMLMessage.
- CorrId - A correlation ID is a user-defined identifier for the message, often used to correlate messages about the same subject
- Priority - An indicator of the level of importance or urgency of the message, with 0 as the lowest priority and 9 as the highest. Usually, 0-4 are gradients of normal priority and 5-9 are gradients of expedited priority. Priority is set to 4 by default.

- Timestamp - The time the message arrived on the queue.

You can change the order in which the columns are listed and choose which of the columns will be included in and which excluded from the display. You can also increase the number of messages displayed on the page from 10 (default) to 20 or 30.

By default, messages are displayed in the order in which they arrived at the destination. You can choose to display the messages in either ascending or descending order by message ID instead by clicking on the ID column header. However, you cannot restore the initial sort order once you have altered it; you must return to the JMS System Module Resources page and reselect the queue to see the messages in order of arrival again.

Querying Messages

The Message Selector field at the top of the JMS Message Management page enables you to filter the messages on the queue based on any valid JMS message header or property with the exception of `JMSXDeliveryCount`. 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 are 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
```

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

Moving Messages

You can move a message from a source destination to a target destination under the following conditions:

- The source destination is either a queue or a topic durable subscriber in the consumption-paused state

  Note: For more information about consumption-paused states, see "Consumption Pause and Consumption Resume" on page 8-20.

- The message state is either visible, delayed, or ordered

- The target destination is
  - in the same cluster as the source destination
  - either a queue, a topic, or a topic durable subscriber
  - not in the production-paused state

  Note: For more information about production-paused states, see "Production Pause and Production Resume" on page 8-16.

The message identifier does not change when you move a message. If the message being moved already exists on the target destination, a duplicate message with the same identifier is added to the destination.

Deleting Messages

You can delete a specific message or drain all messages from a queue under the following conditions:

- The destination is in the consumption-paused state

  Note: For more information about consumption-paused states, see "Consumption Pause and Consumption Resume" on page 8-20.

- The message state is either visible, delayed, or ordered

The destination is locked while the delete operation occurs. If there is a failure during the delete operation, it is possible that only a portion of the messages selected will be deleted.

Creating New Messages

You can create new messages to be sent to a destination. To produce a new message, provide the following information:

- Message type – such as BytesMessage, TextMessage, StreamMessage, ObjectMessage, MapMessage, or XMLMessage.
- Correlation ID – a user-defined identifier for the message, often used to correlate messages about the same subject.
- Expiration – specifies the expiration, or time-to-live value, for a message.
- Priority – an indicator of the level of importance or urgency of the message, with 0 as the lowest priority and 9 as the highest. Usually, 0-4 are gradients of normal priority and 5-9 are gradients of expedited priority. Priority is set to 4 by default.

- Delivery Mode – specifies `PERSISTENT` or `NON_PERSISTENT` messaging.
- Delivery Time – defines the earliest absolute time at which a message can be delivered to a consumer.
- Redelivery Limit – the number of redelivery tries a message can have before it is moved to an error destination.
- Header – 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.
- Body – the message content.

For more information on JMS message properties, see "Understanding WebLogic JMS" in *Programming WebLogic JMS*.

Importing Messages

Importing a message in XML format results in the creation or replacement of a message on the specified destination. The target destination for an imported message can be either a queue or a topic durable subscriber. The destination must be in a production-paused state.

Note: For more information about production-paused states, see "Production Pause and Production Resume" on page 8-16.

If a message being replaced with an imported file is associated with a JMS transaction, the imported replacement will still be associated with the transaction.

When a new message is created or an existing message is replaced with an imported file, the following rules apply:

- Quota limits are enforced for both new messages and replacement messages
- The delivery count of the imported message is set to zero
- A new message ID is generated for each imported message
- If the imported message specifies a delivery mode of `PERSISTENT` and the target destination has no store, the delivery mode is changed to `NON-PERSISTENT`

Note: While importing a JMS message is similar in result to creating or publishing a new JMS message, messages with a defined (non-zero) `ExpirationTime` behave differently when imported, but since the message management API's `ExpirationTime` is absolute for imported messages. Whereas, the message send API's `ExpirationTime` is relative to the time the message is sent.

Exporting Messages

Exporting a message results in a JMS message that is converted to either XML or serialized format. The source destination must be in a production-paused state.

Note: For more information about production-paused states, see "Production Pause and Production Resume" on page 8-16.

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.

Caution: Generally, JMS applications can use the `JMSReplyTo` header field to return a response to a request. However, the information in the `JMSReplyTo` field is not a usable destination object and will not be valid following export or import.

Managing Transactions

When a message is produced or consumed as part of a global transaction, the message is essentially locked by the transaction and will remain locked until the transaction coordinator either commits or aborts the JMS branch. If the coordinator is not able to communicate the outcome of the transaction to the JMS server due to a failure, the message(s) associated with the transaction may remain pending for a long time.

The JMS server transaction management features available through the Administration Console allow you to:

- Identify in-progress transactions for which a JMS server is a participant
- Identify messages associated with a JMS transaction branch
- Force the outcome of pending JMS transaction branches, either by committing them or rolling them back
- Managing JMS Client Connections

You can view all the JMS connections on a particular WebLogic Server instance and get address and port information for each process that is holding a connection. You can also terminate a connection. For more information on using the Administration Console to manage transactions for a JMS server, see "JMS Servers: Monitoring: Active Transactions" in the *Administration Console Online Help*.

For more information on JMS transactions, see "Using Transactions with WebLogic JMS" in *Programming WebLogic JMS*.

Troubleshooting WebLogic JMS

This release of WebLogic Server includes the WebLogic Diagnostic Service, which is a monitoring and diagnostic service that runs within the WebLogic Server process and participates in the standard server life cycle. This service enables you to create, collect, analyze, archive, and access diagnostic data generated by a running server and the applications deployed within its containers. This data provides insight into the runtime performance of servers and applications and enables you to isolate and diagnose faults when they occur. WebLogic JMS takes advantage of this service to provide enhanced runtime statistics, notifications sent to queues and topics, message life cycle logging, and debugging to help you keep your WebLogic domain running smoothly.

For more information on monitoring JMS statistics and managing JMS messages, see "Monitoring JMS Statistics and Managing Messages" on page 7-1.

The following sections explain how to troubleshoot WebLogic JMS messages and configurations:

- "Configuring Notifications for JMS" on page 8-2
- "Debugging JMS" on page 8-2
- "Message Life Cycle Logging" on page 8-7
- "JMS Message Log Content" on page 8-9
- "Controlling Message Operations on Destinations" on page 8-15

Configuring Notifications for JMS

A notification is an action that is triggered when a watch rule evaluates to `true`. JMS notifications are used to post messages to JMS topics and/or queues in response to the triggering of an associated watch. In the system resource configuration file, the elements `<destination-jndi-name>` and `<connection-factory-jndi-name>` define how the message is to be delivered.

For more information, see Configuring Notifications in *Configuring and Using WebLogic Diagnostic Framework*.

Debugging JMS

Once you have narrowed the problem down to a specific application, you can activate WebLogic Server's debugging features to track down the specific problem within the application.

Enabling Debugging

You can enable debugging by setting the appropriate `ServerDebug` configuration attribute to `true`. Optionally, you can also set the server `StdoutSeverity` to `Debug`.

You can modify the configuration attribute in any of the following ways.

Enable Debugging Using the Command Line

Set the appropriate properties on the command line. For example,

```
-Dweblogic.debug.DebugJMSBackEnd=true
-Dweblogic.log.StdoutSeverity="Debug"
```

This method is static and can only be used at server startup.

Enable Debugging Using the WebLogic Server Administration Console

Use the WebLogic Server Administration Console to set the debugging values:

1. If you have not already done so, in the Change Center of the Administration Console, click Lock & Edit (see Use the Change Center).
2. In the left pane of the console, expand Environment and select Servers.
3. On the Summary of Servers page, click the server on which you want to enable or disable debugging to open the settings page for that server.

4. Click Debug.
5. Expand default.
6. Select the check box for the debug scopes or attributes you want to modify.
7. Select Enable to enable (or Disable to disable) the debug scopes or attributes you have checked.
8. To activate these changes, in the Change Center of the Administration Console, click Activate Changes.
   Not all changes take effect immediately—some require a restart (see Use the Change Center).

   This method is dynamic and can be used to enable debugging while the server is running.

Enable Debugging Using the WebLogic Scripting Tool

Use the WebLogic Scripting Tool (WLST) to set the debugging values. For example, the following command runs a program for setting debugging values called `debug.py`:

```
java weblogic.WLST debug.py
```

The main scope, weblogic, does not appear in the graphic; jms is a sub-scope within weblogic. Note that the fully-qualified `DebugScope` for `DebugJMSBackEnd` is `weblogic.jms.backend`.

The debug.py program contains the following code:

```
user='user1'
password='password'
url='t3://localhost:7001'
connect(user, password, url)
edit()
cd('Servers/myserver/ServerDebug/myserver')
startEdit()
set('DebugJMSBackEnd','true')
save()
activate()
```

Note that you can also use WLST from Java. The following example shows a Java file used to set debugging values:

```
import weblogic.management.scripting.utils.WLSTInterpreter;
import java.io.*;
import weblogic.jndi.Environment;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class test {
        public static void main(String args[]) {
       try {
              WLSTInterpreter interpreter = null;
              String user="user1";
              String pass="pw12ab";
              String url ="t3://localhost:7001";
              Environment env = new Environment();
              env.setProviderUrl(url);
              env.setSecurityPrincipal(user);
              env.setSecurityCredentials(pass);
              Context ctx = env.getInitialContext();

              interpreter = new WLSTInterpreter();
              interpreter.exec
                     ("connect('"+user+"','"+pass+"','"+url+"')");
              interpreter.exec("edit()");
              interpreter.exec("startEdit()");
              interpreter.exec
                     ("cd('Servers/myserver/ServerDebug/myserver')");
              interpreter.exec("set('DebugJMSBackEnd','true')");
              interpreter.exec("save()");
              interpreter.exec("activate()");

       } catch (Exception e) {
       System.out.println("Exception "+e);
       }
       }
}
```

Using the WLST is a dynamic method and can be used to enable debugging while the server is running.

Changes to the config.xml File

Changes in debugging characteristics, through console, or WLST, or command line are persisted in the `config.xml` file.

This sample `config.xml` fragment shows a transaction debug scope (set of debug attributes) and a single JMS attribute.

Listing 8-1 Example Debugging Stanza for JMS

```
<server>
<name>myserver</name>
<server-debug>
<debug-scope>
<name>weblogic.transaction</name>
<enabled>true</enabled>
</debug-scope>
<debug-jms-back-end>true</debug-jms-back-end>
</server-debug>
</server>
```

JMS Debugging Scopes

It is possible to see the tree view of the DebugScope definitions using `java weblogic.diagnostics.debug.DebugScopeViewer.`

You can enable the following registered debugging scopes for JMS:

- DebugJMSBackEnd (scope `weblogic.jms.backend`) - prints information for debugging the JMS Back End (including some information used for distributed destinations and JMS SAF).
- DebugJMSFrontEnd (scope `weblogic.jms.frontend`) - prints information for debugging the JMS Front End (including some information used for multicast).
- DebugJMSCommon (scope `weblogic.jms.common`) - prints information for debugging JMS common methods (including some information from the client JMS producer).
- DebugJMSConfig (scope `weblogic.jms.config`) - prints information related to JMS configuration (backend, distributed destinations, and foreign servers).

- DebugJMSBoot (scope `weblogic.jms.boot`) - prints a few messages a boot time regarding what store the JMS server is using and configured destinations.
- DebugJMSDispatcher (scope `weblogic.jms.dispatcher`) - prints information related to `PeerGone()` occurrences.
- DebugJMSDistTopic (scope `weblogic.jms.config`) - prints information about distributed topics, primary bind and unbind information.
- DebugJMSPauseResume (scope `weblogic.jms.pauseresume`) - prints information about (backend) pause/resume operations.
- DebugJMSModule (scope `weblogic.jms.module`) - prints a lot of information about JMS module operations and life cycle.
- DebugJMSMessagePath (scope `weblogic.jms.messagePath`) - prints information following a message through the message path (client, frontend, backend), including the message identifier.
- DebugJMSSAF (scope `weblogic.jms.saf`) - prints information about JMS SAF.
- DebugJMSCDS (scope `weblogic.jms.CDS`) - prints detailed information about JMS "Configuration Directory Service" (used by various sub-systems to get the notification of configuration changes to the JMS resources configured in the server from within a cluster as well as across the clusters and domains).
- DebugJMSWrappers (scope `weblogic.jms.wrappers`) - prints information pooling and wrapping of JMS connections, sessions, and other objects, used inside an EJB or servlet using the `resource-reference` element in the deployment descriptor.

Request Dyeing

Another option for debugging is to trace the flow of an individual (typically "dyed") application request through the JMS subsystem. For more information, see "Configuring the Dye Vector via the DyeInjection Monitor" in *Configuring and Using the WebLogic Diagnostic Framework*.

Message Life Cycle Logging

JMS logging is enabled by default when you create a JMS server, however, you must specifically enable it on message destinations in the JMS modules targeted to this JMS server (or on the JMS template used by destinations). For more information on WebLogic logging services, see "Understanding WebLogic Logging Services" in *Configuring Log Files and Filtering Log Messages*.

The message life cycle is an external view of the events that a JMS message traverses through once it has been accepted by the JMS server, either through the JMS APIs or the JMS Message Management APIs. Message life cycle logging provides an administrator with easy access to information about the existence and status of JMS messages from the JMS server viewpoint. In particular, each message log contains information about basic life cycle events such as message production, consumption, and removal.

Logging can occur on a continuous basis and over a long period of time. It can be also be used in real-time mode while the JMS server is running, or in an off-line fashion when the JMS server is down. For information about configuring message logging, see the following sources.

- "View and configure logs" in the *Administration Console Online Help*
- "Configure JMS server message log rotation" in the *Administration Console Online Help*
- "Configure topic message logging" in the *Administration Console Online Help*
- "Configure queue message logging" in the *Administration Console Online Help*
- "Configure JMS template message logging" in the *Administration Console Online Help*
- "Configure uniform distributed topic message logging" in the *Administration Console Online Help*
- "Configure uniform distributed queue message logging" in the *Administration Console Online Help*

Events in the JMS Message Life Cycle

When message life cycle logging is enabled for a JMS destination, a record is added to the JMS server's message log file each time a message meets the conditions that correspond to a basic message life cycle event. The life cycle events that trigger a JMS message log entry are as follows:

- Produced – This event is logged when a message enters a JMS server via the WebLogic Server JMS API or the JMS Management API.
- Consumed – This event is logged when a message leaves a JMS server via the WebLogic Server JMS API or the JMS Management API.
- Removed – This event is logged when a message is manually deleted from a JMS server via the WebLogic Server JMS API or the JMS Management API.
- Expired – This event is logged when a message reaches the expiration time stored on the JMS server. This event is logged only once per message even though a separate expiration event occurs for each topic subscriber who received the message.
- Retry exceeded – This event is logged when a message has exceeded its redelivery retry limit. This event may be logged more than one time per message, as each topic subscriber has its own redelivery count.
- Consumer created – This event is logged when a JMS consumer is created for a queue or a JMS durable subscriber is created for a topic.
- Consumer destroyed – This event is logged when a JMS consumer is closed or a JMS durable subscriber is unsubscribed.

Message Log Location

The message log is stored under your domain directory, as follows:

```
USER_DOMAIN\servers\servername\logs\jmsServers\jms_server_name\jms.message
s.log
```

where *`USER_DOMAIN`* is the root directory of your domain, typically `c:\bea\user_projects\domains\`*`USER_DOMAIN`*, which is parallel to the directory in which WebLogic Server program files are stored, typically `c:\bea\weblogic90`.

Enabling JMS Message Logging

You can enable or disable JMS message logging for a queue, topic, JMS template, uniform distributed queue, and uniform distributed topic using the WebLogic Server Administration Console. For more information see the following sources.

- "Configure topic message logging" in the *Administration Console Online Help*
- "Configure queue message logging" in the *Administration Console Online Help*
- "Configure JMS template message logging" in the *Administration Console Online Help*

- "Configure uniform distributed topic message logging" in the *Administration Console Online Help*
- "Configure uniform distributed queue message logging" in the *Administration Console Online Help*

WebLogic Java Management Extensions (JMX) enables you to access the JMSSystemResourceMBean and JMSRuntimeMBean MBeans to manage JMS message logs. For more information see "Overview of WebLogic Server Subsystem MBeans" in *Programming WebLogic Management Services with JMX*

You can also use the WebLogic Scripting Tool to configure JMS message logging for a JMS servers and JMS system resources. For more information, see Chapter 6, "Using WLST to Manage JMS Servers and JMS System Resources.".

When you enable message logging, you can specify whether the log entry will include all the message header fields or a subset of them; all system-defined message properties or a subset of them; all user-defined properties or a subset of them. You may also choose to include or to exclude the body of the message. For more information about message headers and properties see "Message Object" in *Programming WebLogic JMS.*

JMS Message Log Content

Each record added to the log includes basic information such as the message ID and correlation ID for the subject message. You can also configure the JMS server to include additional information such as the message type and user properties.

JMS Message Log Record Format

Except where noted, all records added to the JMS Message Life Cycle Log contain the following pieces of information in the order in which they are listed:

- Date – The date and time the message log record is generated.
- Transaction identifier – The transaction identifier for the transaction with which the message is associated
- WLS diagnostic context – A unique identifier for a request or unit of work flowing through the system. It is included in the JMS message log to provide a correlation between events belonging to the same request.
- Raw millisecond value for "Date" – To aid in troubleshooting high-traffic applications, the date and time the message log record is generated is displayed in milliseconds.

- Raw nanosecond value for “Date” – To aid in troubleshooting high-traffic applications, the date and time the message log record is generated is displayed in nanoseconds.
- JMS message ID – The unique identifier assigned to the message.
- JMS correlation ID – A user-defined identifier for the message, often used to correlate messages about the same subject.
- JMS destination name – The fully-qualified name of the destination server for the message.
- JMS message life cycle event name – The name of the message life cycle event that triggered the log entry.
- JMS user name – The name of the user who (produced? consumed? received?) the message.
- JMS message consumer identifier – This information is included in the log only when the message life cycle event being logged is the “Consumed” event, the “Consumer Created” event, or the “Consumer Destroyed” event. If the message consumed was on a queue, the log will include information about the origin of the consumer and the OAM identifier for the consumer known to the JMS server. If the consumer is a durable subscriber, the log will also include the client ID for the connection and the subscription name.

  The syntax for the message consumer identifier is as follows:

  ```
  MC:CA(…):OAMI(wls_server_name.jms.connection#.session#.consumer#)
  ```

  where

  - MC stands for message consumer,
  - CA stands for client address,
  - OAMI stands for OA&M identifier,
  - and, when applicable, CC stands for connection consumer.

  If the consumer is a durable subscriber the additional information will be shown using the following syntax:

  ```
  DS:client_id.subscription_name[message consumer identifier]
  ```

  where DS stands for durable subscriber.
- JMS message content – This field can be customized on a per destination basis. However, the message body will not be available.

- JMS message selector – This information is included in the log only when the message life cycle event being logged is the “Consumer Created” event. The log will show the “Selector” argument from the JMS API.

Sample Log File Records

The sample log file records that follow show the type of information that is provided in the log file for each of the message life cycle events. Each record is a fixed length, but the information included will vary depending upon relevance to the event and on whether a valid value exists for each field in the record. The log file records use the following syntax:

```
####<date_and_time_stamp> <transaction_id> <WLS_diagnostic_context>
<date_in_milliseconds> <date_in_nanoseconds> <JMS_message_id>
<JMS_correlation_id> <JMS_destination_name> <life_cycle_event_name>
<JMS_user_name> <consumer_identifier> <JMS_message_content>
<JMS_message_selector>
```

Note: If you choose to include the JMS message content in the log file, note that any occurrences of the left-pointing angle bracket (<) and the right-pointing angle bracket (>) within the contents of the message will be escaped. In place of a left-pointing angle bracket you will see the string “&lt;” and in place of the right-pointing angle bracket you will see “&gt;” in the log file.

Consumer Created Event

```
####<May 13, 2005 4:06:33 PM EDT> <> <> <1116014793818> <345063> <> <>
<jmsfunc!TestQueueLogging> <ConsumerCreate> <system>
<MC:CA(/10.61.6.56):OAMI(myserver.jms.connection456.session460.consumer462
)> <> <>
```

```
Consumer Destroyed Event
####<May 13, 2005 4:06:33 PM EDT> <> <> <1116014793844> <40852> <> <>
<jmsfunc!TestQueueLogging> <ConsumerDestroy> <system>
<MC:CA(/10.61.6.56):OAMI(myserver.jms.connection456.session460.consumer462
)> <> <>
```

Message Produced Event

```
####<May 13, 2005 4:06:43 PM EDT> <> <> <1116014803018> <693671>
<ID:<327315.1116014803000.0>> <testSendRecord>
```

```
<jmsfunc!TestQueueLoggingMarker> <Produced> <system> <> <&lt;?xml
version="1.0" encoding="UTF-8"?&gt;

&lt;mes:WLJMSMessage
xmlns:mes="http://www.bea.com/WLS/JMS/Message"&gt;&lt;mes:Header&gt;&lt;me
s:JMSCorrelationID&gt;testSendRecord&lt;/mes:JMSCorrelationID&gt;&lt;mes:J
MSDeliveryMode&gt;NON_PERSISTENT&lt;/mes:JMSDeliveryMode&gt;&lt;mes:JMSExp
iration&gt;0&lt;/mes:JMSExpiration&gt;&lt;mes:JMSPriority&gt;4&lt;/mes:JMS
Priority&gt;&lt;mes:JMSRedelivered&gt;false&lt;/mes:JMSRedelivered&gt;&lt;
mes:JMSTimestamp&gt;1116014803000&lt;/mes:JMSTimestamp&gt;&lt;mes:Properti
es&gt;&lt;mes:property
name="JMSXDeliveryCount"&gt;&lt;mes:Int&gt;0&lt;/mes:Int&gt;&lt;/mes:prope
rty&gt;&lt;/mes:Properties&gt;&lt;/mes:Header&gt;&lt;mes:Body&gt;&lt;mes:T
ext/&gt;&lt;/mes:Body&gt;&lt;/mes:WLJMSMessage&gt;> <>
```

Message Consumed Event

```
####<May 13, 2005 4:06:45 PM EDT> <> <> <1116014805137> <268791>
<ID:<327315.1116014804578.0>> <hello> <jmsfunc!TestQueueLogging>
<Consumed> <system>
<MC:CA(/10.61.6.56):OAMI(myserver.jms.connection456.session475.consumer477
)> <&lt;?xml version="1.0" encoding="UTF-8"?&gt;

&lt;mes:WLJMSMessage
xmlns:mes="http://www.bea.com/WLS/JMS/Message"&gt;&lt;mes:Header&gt;&lt;me
s:JMSCorrelationID&gt;hello&lt;/mes:JMSCorrelationID&gt;&lt;mes:JMSDeliver
yMode&gt;PERSISTENT&lt;/mes:JMSDeliveryMode&gt;&lt;mes:JMSExpiration&gt;0&
lt;/mes:JMSExpiration&gt;&lt;mes:JMSPriority&gt;4&lt;/mes:JMSPriority&gt;&
lt;mes:JMSRedelivered&gt;false&lt;/mes:JMSRedelivered&gt;&lt;mes:JMSTimest
amp&gt;1116014804578&lt;/mes:JMSTimestamp&gt;&lt;mes:JMSType&gt;SendRecord
&lt;/mes:JMSType&gt;&lt;mes:Properties&gt;&lt;mes:property
name="JMS_BEA_RedeliveryLimit"&gt;&lt;mes:Int&gt;1&lt;/mes:Int&gt;&lt;/mes
:property&gt;&lt;mes:property
name="JMSXDeliveryCount"&gt;&lt;mes:Int&gt;1&lt;/mes:Int&gt;&lt;/mes:prope
rty&gt;&lt;/mes:Properties&gt;&lt;/mes:Header&gt;&lt;mes:Body&gt;&lt;mes:T
ext/&gt;&lt;/mes:Body&gt;&lt;/mes:WLJMSMessage&gt;> <>
```

Message Expired Event

```
####<May 13, 2005 4:06:47 PM EDT> <> <> <1116014807258> <445317>
<ID:<327315.1116014807234.0>> <bar> <jmsfunc!TestQueueLogging> <Expired>
<<WLS Kernel>> <> <&lt;?xml version="1.0" encoding="UTF-8"?&gt;

&lt;mes:WLJMSMessage
xmlns:mes="http://www.bea.com/WLS/JMS/Message"&gt;&lt;mes:Header&gt;&lt;me
s:JMSCorrelationID&gt;bar&lt;/mes:JMSCorrelationID&gt;&lt;mes:JMSDeliveryM
ode&gt;PERSISTENT&lt;/mes:JMSDeliveryMode&gt;&lt;mes:JMSExpiration&gt;1116
014806234&lt;/mes:JMSExpiration&gt;&lt;mes:JMSPriority&gt;4&lt;/mes:JMSPri
ority&gt;&lt;mes:JMSRedelivered&gt;false&lt;/mes:JMSRedelivered&gt;&lt;mes
:JMSTimestamp&gt;1116014807234&lt;/mes:JMSTimestamp&gt;&lt;mes:JMSType&gt;
ExpireRecord&lt;/mes:JMSType&gt;&lt;mes:Properties&gt;&lt;mes:property
name="JMS_BEA_RedeliveryLimit"&gt;&lt;mes:Int&gt;1&lt;/mes:Int&gt;&lt;/mes
:property&gt;&lt;mes:property
name="JMSXDeliveryCount"&gt;&lt;mes:Int&gt;0&lt;/mes:Int&gt;&lt;/mes:prope
rty&gt;&lt;/mes:Properties&gt;&lt;/mes:Header&gt;&lt;mes:Body&gt;&lt;mes:T
ext/&gt;&lt;/mes:Body&gt;&lt;/mes:WLJMSMessage&gt;> <>
```

Retry Exceeded Event

```
####<May 13, 2005 4:06:53 PM EDT> <> <> <1116014813491> <394206>
<ID:<327315.1116014813453.0>> <bar> <jmsfunc!TestQueueLogging> <Retry
exceeded> <<WLS Kernel>> <> <&lt;?xml version="1.0" encoding="UTF-8"?&gt;

&lt;mes:WLJMSMessage
xmlns:mes="http://www.bea.com/WLS/JMS/Message"&gt;&lt;mes:Header&gt;&lt;me
s:JMSCorrelationID&gt;bar&lt;/mes:JMSCorrelationID&gt;&lt;mes:JMSDeliveryM
ode&gt;PERSISTENT&lt;/mes:JMSDeliveryMode&gt;&lt;mes:JMSExpiration&gt;0&lt
;/mes:JMSExpiration&gt;&lt;mes:JMSPriority&gt;4&lt;/mes:JMSPriority&gt;&lt
;mes:JMSRedelivered&gt;true&lt;/mes:JMSRedelivered&gt;&lt;mes:JMSTimestamp
&gt;1116014813453&lt;/mes:JMSTimestamp&gt;&lt;mes:JMSType&gt;RetryRecord&l
t;/mes:JMSType&gt;&lt;mes:Properties&gt;&lt;mes:property
name="JMS_BEA_RedeliveryLimit"&gt;&lt;mes:Int&gt;1&lt;/mes:Int&gt;&lt;/mes
:property&gt;&lt;mes:property
name="JMSXDeliveryCount"&gt;&lt;mes:Int&gt;2&lt;/mes:Int&gt;&lt;/mes:prope
rty&gt;&lt;/mes:Properties&gt;&lt;/mes:Header&gt;&lt;mes:Body&gt;&lt;mes:T
ext/&gt;&lt;/mes:Body&gt;&lt;/mes:WLJMSMessage&gt;> <>
```

Message Removed Event

```
####<May 13, 2005 4:06:45 PM EDT> <> <> <1116014805071> <169809>
<ID:<327315.1116014804859.0>> <hello> <jmsfunc!TestTopicLogging> <Removed>
<system> <DS:messagelogging_client.foo.SendRecordSubscriber> <&lt;?xml
version="1.0" encoding="UTF-8"?&gt;

&lt;mes:WLJMSMessage
xmlns:mes="http://www.bea.com/WLS/JMS/Message"&gt;&lt;mes:Header&gt;&lt;me
s:JMSCorrelationID&gt;hello&lt;/mes:JMSCorrelationID&gt;&lt;mes:JMSDeliver
yMode&gt;PERSISTENT&lt;/mes:JMSDeliveryMode&gt;&lt;mes:JMSExpiration&gt;0&
lt;/mes:JMSExpiration&gt;&lt;mes:JMSPriority&gt;4&lt;/mes:JMSPriority&gt;&
lt;mes:JMSRedelivered&gt;false&lt;/mes:JMSRedelivered&gt;&lt;mes:JMSTimest
amp&gt;1116014804859&lt;/mes:JMSTimestamp&gt;&lt;mes:JMSType&gt;SendRecord
Subscriber&lt;/mes:JMSType&gt;&lt;mes:Properties&gt;&lt;mes:property
name="JMSXDeliveryCount"&gt;&lt;mes:Int&gt;0&lt;/mes:Int&gt;&lt;/mes:prope
rty&gt;&lt;/mes:Properties&gt;&lt;/mes:Header&gt;&lt;mes:Body&gt;&lt;mes:T
ext/&gt;&lt;/mes:Body&gt;&lt;/mes:WLJMSMessage&gt;> <>
```

Managing JMS Server Log Files

After you create a JMS server, you can configure criteria for moving (rotating) old log messages to a separate file. You can also change the default name of the log file.

Rotating Message Log Files

You can choose to rotate old log messages to a new file based on a specific file size or at specified intervals of time. Alternately, you can choose not to rotate old log messages; in this case, all messages will accumulate in a single file and you will have to erase the contents of the file when it becomes too large.

If you choose to rotate old messages whenever the log file reaches a particular size you must specify a minimum file size. After the log file reaches the specified minimum size, the next time the server checks the file size it will rename the current log file and create a new one for storing subsequent messages.

If you choose to rotate old messages at a regular interval, you must specify the time at which the first new message log file is to be created, and then specify the time interval that should pass before that file is renamed and replaced.

For more information about setting up log file rotation for JMS servers, see “Configure JMS server message log rotation” in the *Administration Console Online Help*.

Renaming Message Log Files

Rotated log files are numbered in order of creation. For example, the seventh rotated file would be named `myserver.log00007`. For troubleshooting purposes, it may be useful to change the name of the log file or to include the time and date when the log file is rotated. To do this, you add `java.text.SimpleDateFormat` variables to the file name. Surround each variable with percentage (`%`) characters. If you specify a relative pathname when you change the name of the log file, it is interpreted as relative to the server's root directory.

For more information about renaming message log files for JMS servers, see "Configure JMS server message log rotation" in the *Administration Console Online Help*.

Limiting the Number of Retained Message Log Files

If you choose to rotate old message log files based on either file size or time interval, you may also wish to limit the number of log files this JMS server creates for storing old messages. After the server reaches this limit, it deletes the oldest log file and creates a new log file with the latest suffix. If you do not enable this option, the server will create new files indefinitely and you will have to manually clean up these files.

For more information about limiting the number of message log files for JMS servers, see "Configure JMS server message log rotation" in the *Administration Console Online Help*.

Controlling Message Operations on Destinations

WebLogic JMS configuration and runtime APIs enable you to pause and resume message production, insertion, and/or consumption operations on a JMS destination or temporary destination, on a group of destinations configured using the same template, or on all the destinations hosted by a single JMS Server, either programmatically (using JMX and the runtime MBean API) or administratively (using the Administration Console). In this way, you can control the JMS subsystem behavior in the event of an external resource failure that would otherwise cause the JMS subsystem to overload the system by continuously accepting and delivering (and redelivering) messages.

You can boot a JMS server and its destinations in a "paused" state which prevents any message production, insertion, or consumption on those destinations immediately after boot. To resume message operation activity, the administrator can later change the state of the paused destination to "resume" normal message production, insertion, or consumption operations. In addition, new runtime options allow an administrator to change the current state of a running destination to either allow or disallow new message production, insertion, or consumption.

Definition of Message Production, Insertion, and Consumption

There are several operations performed on messages on a destination:

- Messages are *produced* when a producer creates and sends a new message to that destination.
- Messages are *inserted* as a result of in-flight work completion, as when a message is made available upon commitment of a transaction or when a message scheduled to be made available after a delay is made available on a destination.
- Messages are *consumed* when they are removed from the destination.

You can pause and resume any or all of these operations either at boot time or during runtime, as described in the sections below.

Pause and Resume Logging

When message production, insertion, or consumption on a destination is successfully "paused" or "resumed" either at boot time or at runtime, a message is added to the server log to indicate the same. In the event of failure to pause or resume message production, insertion, or consumption on a destination, the appropriate error/exceptions are logged.

Production Pause and Production Resume

When a JMS destination is "paused for production," new and existing producers attached to that destination are unable to produce new messages for that destination. A producer that attempts to send a message to a paused destination receives an exception that indicates that the destination is paused. When a destination is "resumed from production pause," production of new messages is allowed again. Pausing message production does not prevent the insertion of messages that are the result in-flight work.

Notes: For an explanation of what constitutes in-flight work, see "Definition of In-Flight Work" on page 8-22.

Pausing and Resuming Production at Boot-time

You can pause or resume production effective at boot-time for all the destinations on a JMS server, for a group of destinations that point to the same JMS template, or for individual destinations. If you configure `production-paused-at-startup`, the next time you boot the server, message production activities will be disallowed for the specified destination(s) until you explicitly change the state to "production enabled" for that destination. If you configure production to resume, the next time you boot the server, message production activities will be allowed on the specified destination(s) until the state is explicitly changed to "production paused" for that destination.

For more information about pausing and resuming message production at boot-time using the Administration console, see the following sources.

- "Pause JMS server message operations on server restart" in the *Administration Console Online Help*
- "Pause topic message operations on server restart" in the *Administration Console Online Help*
- "Pause queue message operations on server restart" in the *Administration Console Online Help*
- "Pause JMS template message operations on server restart" in the *Administration Console Online Help*
- "Pause uniform distributed topic message operations on server restart" in the *Administration Console Online Help*
- "Pause uniform distributed queue message operations on server restart" in the *Administration Console Online Help*

Note: Because it is possible that this operation may be configured differently at each level (i.e., the JMS Server level, the JMS template level, and the standalone destination or uniform distributed destination level), there is an established order of precedence. For more information, see "Order of Precedence for Boot-time Pause and Resume of Message Operations" on page 8-24.

Pausing and Resuming Production at Runtime

You can pause or resume production during runtime for all the destinations targeted on a JMS server, for a group of destinations that point to the same JMS template, or for individual destinations. The most recent configuration change always take precedence, regardless of the level at which it is made (JMS server level, JMS template level, or destination level).

For more information about pausing and resuming production at runtime, see the following sources.

- "Pause JMS server message operations at runtime" in the *Administration Console Online Help*
- "Pause topic message operations at runtime" in the *Administration Console Online Help*
- "Pause queue message operations at runtime" in the *Administration Console Online Help*

Production Pause and Resume and Distributed Destinations

If a member destination is paused for production, that member destination will not be considered for production by the producer. Messages will be steered away to other member destinations that are available for production.

Production Pause and Resume and JMS Connection Stop/Start

Stopping or starting a JMS connection has no effect on the production pause or production resume state of a destination.

Insertion Pause and Insertion Resume

When a JMS destination is paused for "insertion," both messages inserted as a result of in-flight work and new messages sent by producers are prevented from appearing on the destination. Use insertion pause to stop all messages from appearing on a destination.

You can determine whether there is any in-flight work pending by looking at the statistics on the Administration Console. When you pause the destination for message "insertion", messages related to in-flight work completion are made "not deliverable" and new message production operations fail. All of those messages become "invisible" to the consumers and the statistics are adjusted to reflect that the messages are no longer pending.

The "insertion" pause operation supersedes the "production" pause operation. In other words, if the destination is currently in the "production paused" state, you can change it to the "insertion paused" state.

You must explicitly "resume" a destination for message insertion to allow in-flight messages to appear on that destination. Successful completion of the insertion "resume" operation will change the state of the destination to "insertion enabled" and all the "invisible" in-flight messages will be made available.

Pausing and Resuming Insertion at Boot Time

You can pause or resume insertion effective at boot-time for all the destinations on a JMS server, for a group of destinations that point to the same JMS template, or for individual destinations. If you configure `insertion-paused-at-startup`, the next time you boot the server, message insertion and production activities will be disallowed on the specified destination(s) until you explicitly change the state to "insertion enabled" for that destination. If you configure insertion to resume, the next time you boot the server, message insertion activities will be allowed on the specified destination(s) until the state is explicitly changed to "insertion paused" for that destination.

For more information about pausing and resuming message insertion at boot-time, see the following sources.

- "Pause JMS server message operations on server restart" in the *Administration Console Online Help*
- "Pause topic message operations on server restart" in the *Administration Console Online Help*
- "Pause queue message operations on server restart" in the *Administration Console Online Help*
- "Pause JMS template message operations on server restart" in the *Administration Console Online Help*
- "Pause uniform distributed topic message operations on server restart" in the *Administration Console Online Help*
- "Pause uniform distributed queue message operations on server restart" in the *Administration Console Online Help*

Note: Because it is possible that this operation may be configured differently at each level (i.e., the JMS Server level, the JMS template level, and the destination level), there is an established order of precedence. For more information, see "Order of Precedence for Boot-time Pause and Resume of Message Operations" on page 8-24.

Pausing and Resuming Insertion at Runtime

You can pause or resume insertion during runtime for all the destinations on a JMS server, for a group of destinations that point to the same JMS template, or for individual destinations. The most recent configuration change always take precedence, regardless of the level at which it is made (JMS Server level, JMS Template level, or destination level).

For more information about pausing and resuming insertion at runtime, see the following sources.

- "Pause JMS server message operations at runtime" in the *Administration Console Online Help*
- "Pause topic message operations at runtime" in the *Administration Console Online Help*
- "Pause queue message operations at runtime" in the *Administration Console Online Help*

Insertion Pause and Resume and Distributed Destination

If a member destination is paused for insertion, that member destination will not be considered for message forwarding. Messages will be steered away to other member destinations that are available for insertion.

Insertion Pause and Resume and JMS Connection Stop/Start

Stopping or starting a JMS Connection has no effect on the insertion pause or insertion resume state of a destination.

Consumption Pause and Consumption Resume

When a JMS destination is "paused for consumption," messages on that destination are not available for consumption. When the destination is "resumed from consumption pause", both new and existing consumers attached to that destination are allowed to consume messages on the destination again.

When the destination is paused for consumption, the destination's state is marked as "consumption paused" and all new, synchronous receive operations will block until consumption is resumed and there are messages available for consumption. All synchronous receive with blocking time-out operations will block for the specified length of time. Messages will not be delivered to synchronous consumers attached to that destination while the destination is paused for consumption.

After a successful consumption "pause" operation, the user has to explicitly "resume" the destination to allow consume operations on that destination.

Pausing and Resuming Consumption at Boot-time

You can pause or resume consumption effective at boot-time for all the destinations on a JMS server, for a group of destinations that point to the same JMS template, or for individual destinations. If you configure `consumption-paused-at-startup`, the next time you boot the server, message consumption activities will be disallowed on the specified destination(s) until you explicitly change the state to "consumption enabled" for that destination. If you configure consumption to resume, the next time you boot the server, message consumption activities will be allowed on the specified destination(s) until the state is explicitly changed to "consumption paused" for that destination.

For more information about pausing and resuming consumption at boot-time, see the following sources.

- "Pause JMS server message operations on server restart" in the *Administration Console Online Help*
- "Pause topic message operations on server restart" in the *Administration Console Online Help*
- "Pause queue message operations on server restart" in the *Administration Console Online Help*
- "Pause JMS template message operations on server restart" in the *Administration Console Online Help*
- "Pause uniform distributed topic message operations on server restart" in the *Administration Console Online Help*
- "Pause uniform distributed queue message operations on server restart" in the *Administration Console Online Help*

Pausing and Resuming Consumption at Runtime

You can pause or resume consumption during runtime for all the destinations on a JMS server, for a group of destinations that point to the same JMS template, or for individual destinations. The most recent configuration change always take precedence, regardless of the level at which it is made (JMS Server level, JMS Template level, or destination level).

For more information about pausing and resuming consumption at runtime, see the following sources.

- "Pause JMS server message operations at runtime" in the *Administration Console Online Help*

- "Pause topic message operations at runtime" in the *Administration Console Online Help*
- "Pause queue message operations at runtime" in the *Administration Console Online Help*

Consumption Pause and Resume and Queue Browsers

Queue Browsers are special type of consumers that are only allowed to "peek" into queue destinations. A browse operation on a destination paused for consumption is perfectly legitimate and is allowed.

Consumption Pause and Resume and Distributed Destination

Member destinations that are currently paused for consumption are not considered by the consumer load balancing algorithm.

Consumption Pause and Resume and Message-Driven Beans

Pausing a destination for consumption will prevent a message-driven bean (MDB) from getting any messages from its associated destination. This feature gives you more flexible control over the delivery of messages delivery to MDBs from the individual destination level as opposed to using connection start/stop. In other words, if you use the consumption pause/resume feature, you can share the JMS connection among the multiple MDBs and still be able to prevent message delivery to selected MDBs by pausing the associated destination for consumption.

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

Consumption Pause and Resume and JMS Connection Stop/Start

The JMS connection stop/start feature determines whether a consumer can successfully invoke the receive APIs or not. The consumption pause/resume feature on a destination determines whether the receive call will get any messages from the destination or not. Stopping or starting a consumer's connection does not have any impact on the destination's consumption pause state.

If the consumer's connection is "started" from the "stopped" state, synchronous receive operations might block or time-out if the destination is currently paused for consumption. Asynchronous consumers will not receive any messages if the associated destination is in "consumption paused" state.

Definition of In-Flight Work

- "In-flight Work Associated with Producers" on page 8-23

- "In-flight Work Associated with Consumers" on page 8-23

In-flight Work Associated with Producers

The following types of messages are inserted on a destination as a result of in-flight work associated with message producers.

- Unborn Messages – Messages that are created by the producer with "birth time" (TimeToDeliver) set in the future. Until delivered, unborn messages are counted as "pending" messages in the destination statistics and are not available for consumption.
- Uncommitted Messages – Messages that are produced as part of a transaction (using either user transaction or transacted session) and have not yet been either committed or rolled back. Until the transaction has been completed, uncommitted messages are counted as "pending" messages in the destination statistics and are not available for consumption.
- Quota Blocking Send – Messages that, if initially prevented from reaching a destination due to a quota limit, will block for a specific period of time while waiting for the destination to become available. The message may exceed the message quota limit, the byte quota limit, or both quota limits on the destination. While blocking, these messages are invisible to the system and are not counted against any of the destination statistics.

In-flight Work Associated with Consumers

The following types of messages are inserted on a destination as a result of in-flight work associated with message consumers.

- Unacknowledged (CLIENT ACK PENDING) Messages – Messages that have been received by a client and are awaiting acknowledgement from the client. These are "pending messages" which are removed from the destination/system when the acknowledgement is received.
- Uncommitted Messages – Messages that have been received by a client within a transaction which has not yet been committed or rolled back. When the client successfully commits the transaction the messages are removed from the system.
- Rolled-back Messages – Messages that are put back on a destination because of the successful rollback of a transaction.

  These messages might or might not be ready for redelivery to the clients immediately, depending on the redelivery parameters (i.e., RedeliveryDelay and/or RedeliveryDelayOverride and RedeliveryLimit) configured on the associated connection factory and destination.

If there is a redelivery delay configured, then, for the duration of that delay, the messages are not available for redelivery and the messages are counted as "pending" in the destination statistics. After the delay period, if the redelivery limit has not been exceeded, then they are delivered and are counted as "current" messages in the destination statistics. If the redelivery limit has been exceeded, then the messages are moved to the error destination, if one has been configured, or are dropped, if no error destination has been configured.

- Recovered Messages – Messages that appear on the queue because of an explicit call to session "recover" by the client. These messages are similar to the Rolled-back Messages discussed above.
- Redelivered Messages – Messages that reappear on the destination because of an unsuccessful delivery attempt to the client. These messages are similar to the Rolled-back Messages discussed above.

Order of Precedence for Boot-time Pause and Resume of Message Operations

You can pause and resume destinations at boot-time by setting attributes at several different levels:

- If you are using a JMS server to host a group of destinations, you can pause or resume message operations on the entire group of destinations.
- If you are using a JMS template to define the attribute values of groups of destinations, you can pause or resume message operations on all of the destinations in a group.
- You can pause and resume message operations on a single destination.

If the values at each of these levels are not in agreement at boot-time, the following order of precedence is used to determine the behavior of the message operations on the specified destination(s). For each of the attributes used to configure pausing and resumption of message operations:

1. If the hosting JMS server for the destination has the attribute set with a valid value, then that value determines the state of the destination at boot time. Server-level settings have first precedence.
2. If the hosting JMS server does not have the attribute set with a valid value, then the value of the attribute on the destination level has second highest precedence and determines the state of the destination at boot time.

3. If neither the hosting JMS server nor the destination has the attribute set with a valid value, then the value of the attribute on the JMS template determines the state of the destination at boot time.
4. If the attribute has not been set at any of the three levels, then the value is assumed to be “false”.

Security

The administrative user/group can override the current state of a destination irrespective of whether the destination's state is currently being controlled by other users.

If two non-administrative users are trying to control the state of the destination, then the following rules apply.

1. Only a user who belongs to the same group as the user who changed the state of the destination to “paused” is allowed to “resume” the destination to the normal operation.
2. If the state change is attempted by two different users who belong to two different groups, the change is not allowed.

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:

Defining Quota

In prior releases, there were multiple levels of quotas: destinations had their own quotas and would also have to compete for quota within a JMS server. In this release, there is only one level of quota: destinations can have their own private quota or they can compete with other destinations using a shared quota.

In addition, a destination that defines its own quota no longer also shares space in the JMS server's quota. Although JMS servers still allow the direct configuration of message and byte quotas, these options are only used to provide quota for destinations that do not refer to a quota resource.

Quota Resources

A quota is a named configurable JMS module resource. It defines a maximum number of messages and bytes, and is then associated with one or more destinations and is responsible for enforcing the defined maximums. Multiple destinations referring to the same quota share available quota according to the sharing policy for that quota resource.

Quota resources include the following configuration parameters:

Table 9-1 Quota Parameters

Attribute	Description
Bytes Maximum and Messages Maximum	The Messages Maximum/Bytes Maximum parameters for a quota resource defines the maximum number of messages and/or bytes allowed for that quota resource. No consideration is given to messages that are pending; that is, messages that are in-flight, delayed, or otherwise inhibited from delivery still count against the message and/or bytes quota.
Quota Sharing	The Shared parameter for a quota resource defines whether multiple destinations referring to the same quota resource compete for resources with each other.
Quota Policy	The Policy parameter defines how individual clients compete for quota when no quota is available. It affects the order in which send requests are unblocked when the Send Timeout feature is enabled on the connection factory, as described in "Defining a Send Timeout on Connection Factories" on page 9-4.

For more information about quota configuration parameters, see QuotaBean in the *WebLogic Server MBean Reference*. For instructions on configuring a quota resource using the Administration Console, see "Create a quota for destinations" in the *Administration Console Online Help*.

Destination-Level Quota

Destinations no longer define byte and messages maximums for quota, but can use a quota resource that defines these values, along with quota policies on sharing and competition.

The Quota parameter of a destination defines which quota resource is used to enforce quota for the destination. This value is dynamic, so it can be changed at any time. However, if there are unsatisfied requests for quota when the quota resource is changed, then those requests will fail with a `javax.jms.ResourceAllocationException`.

Note: Outstanding requests for quota will fail at such time that the quota resource is changed. This does not mean changes to the message and byte attributes for the quota resource, but when a destination switches to a different quota.

JMS Server-Level Quota

In some cases, there will be destinations that do not configure quotas. JMS Server quotas allow JMS servers to limit the resources used by these "quota-less" destinations. All destinations that do not explicitly set a value for the Quota attribute share the quota of the JMS server where they are deployed. The behavior is exactly the same as if there were a special Quota resource defined for each JMS server with the Shared parameter enabled.

The interfaces for the JMS server quota are unchanged from prior releases. The JMS server quota is entirely controlled using methods on the JMSServerMBean. The quota policy for the JMS server quota is set by the Blocking Send Policy parameter on a JMS server, as explained in "Specifying a Blocking Send Policy on JMS Servers" on page 9-3. It behaves just like the Policy setting of any other quota.

Specifying a Blocking Send Policy on JMS Servers

The Blocking Send policies enable you to define the JMS server's blocking behavior on whether to deliver smaller messages before larger ones when multiple message producers are competing for space on a destination that has exceeded its message quota.

To use the Administration Console to define how a JMS server will block message requests when its destinations are at maximum quota.

1. Follow the directions for navigating to the JMS Server: Configuration: Thresholds and Quotas page of the Administration Console in "Configure JMS server thresholds and quota" in the *Administration Console Online Help*.
2. From the Blocking Send Policy list box, select one of the following options:
   - FIFO — All send requests for the same destination are queued up one behind the other until space is available. No send request is permitted to complete when there another send request is waiting for space before it.
   - Preemptive — A send operation can preempt other blocking send operations if space is available. That is, if there is sufficient space for the current request, then that space is used even if there are previous requests waiting for space.

For more information about the Blocking Send Policy field, see "JMS Server: Configuration: Thresholds and Quota" in the *Administration Console Online Help*.

3. Click Save.

Defining a Send Timeout on Connection Factories

The Send Timeout feature provides more control over message send operations by giving message produces the option of waiting a specified length of time until space becomes available on a destination. For example, if a producer makes a request and there is insufficient space, then the producer is blocked until space becomes available, or the operation times out.

To use the Administration Console to define how long a JMS connection factory will block message requests when a destination exceeds its maximum quota.

1. Follow the directions for navigating to the JMS Connection Factory: Configuration: Flow Control page in "Configure message flow control" in the *Administration Console Online Help*.
2. In the Send Timeout field, enter the amount of time, in milliseconds, a sender will block messages when there is insufficient space on the message destination. Once the specified waiting period ends, one of the following results will occur:
   - If sufficient space becomes available before the timeout period ends, the operation continues.
   - If sufficient space does not become available before the timeout period ends, you receive a "resource allocation" exception.

   If you choose not to enable the blocking send policy by setting this value to 0, then you will receive a "resource allocation" exception whenever sufficient space is not available on the destination.

   For more information about the Send Timeout field, see "JMS Connection Factory: Configuration: Flow Control" in the *Administration Console Online Help*.
3. Click Save.

Compressing Messages

To improve the performance of sending messages traveling across JVM boundaries and help conserve disk space, you can specify the automatic compression of any messages that exceed a user-specified threshold size. Message compression can help reduce network bottlenecks by automatically reducing the size of messages sent across network wires. Compressing messages can also conserve disk space when storing persistent messages in file stores or databases.

Note: Compressed messages may actually inadvertently affect destination quotas since some message types actually grow larger when compressed.

A message compression threshold can be set programmatically using a JMS API extension to the `WLMessageProducer` interface, or administratively by either specifying a Default Compression Threshold value on a connection factory or on a JMS SAF remote context.

For instructions on configuring default compression thresholds using the Administration Console, see:

- Connection factories — "Configure default delivery parameters" in the *Administration Console Online Help*.
- Store-and-Forward (SAF) remote contexts — "Configure SAF remote contexts" in the *Administration Console Online Help*.

Once configured, message compression is triggered on producers for client sends, on connection factories for message receives and message browsing, or through SAF forwarding. Messages are compressed using GZIP. Compression only occurs when message producers and consumers are located on separate server instances where messages must cross a JVM boundary, typically across a network connection when WebLogic domains reside on different machines. Decompression automatically occurs on the client side and only when the message content is accessed, except for the following situations:

- Using message selectors on compressed XML messages can cause decompression, since the message body must be accessed in order to filter them. For more information on defining XML message selectors, see "Filtering Messages" in *Programming WebLogic JMS*.
- Interoperating with earlier versions of WebLogic Server can cause decompression. For example, when using the Messaging Bridge, messages are decompressed when sent from the current release of WebLogic Server to a receiving side that is an earlier version of WebLogic Server.

On the server side, messages always remains compressed, even when they are written to disk.

Paging Out Messages To Free Up Memory

With the *message paging* feature, JMS servers automatically attempt to free up virtual memory during peak message load periods. This feature can greatly benefit applications with large message spaces. Message paging is always enabled on JMS servers, and so a message paging directory is automatically created without having to configure one. You can, however, specify a directory using the Paging Directory option, then paged-out messages are written to files in this directory.

JMS message paging saves memory for persistent messages, as even persistent messages cache their data in memory. If a JMS server is associated with a file store (either user-defined or the server's default store), paged persistent messages are generally written to that file store, while non-persistent messages are always written to the JMS server's paging directory. If a JMS server is associated with a JDBC store, then both paged persistent and non-persistent messages are always written to the JMS server's paging directory.

However, a paged-out message does not free all of the memory that it consumes, since the message header -- with the exception of any user properties, which are paged out along with the message body -- remains in memory for use with searching, sorting, and filtering.

Note: Messages that have been received by a consumer, but which have not been acknowledged, are only eligible for paging *after* the session is committed. Prior to that, the message will only be held in memory. Therefore, the heap size of the Java Virtual Machine (JVM) should be appropriately tuned to accommodate the projected peak amount of client load from all active sessions until they are committed.

Specifying a Message Paging Directory

If a paging directory is not specified, then paged-out message bodies are written to the default `\tmp` directory inside the *servername* subdirectory of a domain's root directory. For example, if no directory name is specified for the default paging directory, it defaults to:

`bea_home\user_projects\domains\domainname\servers\servername\tmp`

where `domainname` is the root directory of your domain, typically `c:\bea\user_projects\domains\domainname`, which is parallel to the directory in which WebLogic Server program files are stored, typically `c:\bea\weblogic90`.

Tuning the Message Buffer Size Option

The Message Buffer Size option specifies the amount of memory that will be used to store message bodies in memory before they are paged out to disk. The default value of Message Buffer

Size is approximately one-third of the maximum heap size for the JVM, or a maximum of 512 megabytes. The larger this parameter is set, the more memory JMS will consume when many messages are waiting on queues or topics. Once this threshold is crossed, JMS may write message bodies to the directory specified by the Paging Directory option in an effort to reduce memory usage below this threshold.

It is important to note that this parameter is not a quota. If the number of messages on the server passes the threshold, the server will write messages to disk and evict them from memory as fast as it can to reduce memory usage, but it will not stop accepting new messages. It is still possible to run out of memory if messages are arriving faster than they can be paged out. Users with high messaging loads who wish to support the highest possible availability should consider setting a quota, or setting a threshold and enabling flow control to reduce memory usage on the server.

Controlling the Flow of Messages on JMS Servers and Destinations

With the Flow Control feature, you can direct a JMS server or destination to slow down message producers when it determines that it is becoming overloaded.

The following sections describe how flow control feature works and how to configure flow control on a connection factory.

- "How Flow Control Works" on page 9-7
- "Configuring Flow Control" on page 9-8
- "Flow Control Thresholds" on page 9-9

How Flow Control Works

Specifically, when either a JMS server or it's destinations exceeds its specified byte or message threshold, it becomes *armed* and instructs producers to limit their message flow (messages per second).

Producers will limit their production rate based on a set of flow control attributes configured for producers via the JMS connection factory. Starting at a specified *flow maximum* number of messages, a producer evaluates whether the server/destination is still armed at prescribed intervals (for example, every 10 seconds for 60 seconds). If at each interval, the server/destination is still armed, then the producer continues to move its rate down to its prescribed *flow minimum* amount.

As producers slow themselves down, the threshold condition gradually corrects itself until the server/destination is *unarmed*. At this point, a producer is allowed to increase its production rate, but not necessarily to the maximum possible rate. In fact, its message flow continues to be controlled (even though the server/destination is no longer armed) until it reaches its prescribed *flow maximum*, at which point it is no longer flow controlled.

Configuring Flow Control

Producers receive a set of flow control attributes from their session, which receives the attributes from the connection, and which receives the attributes from the connection factory. These attributes allow the producer to adjust its message flow.

Specifically, the producer receives attributes that limit its flow within a minimum and maximum range. As conditions worsen, the producer moves toward the minimum; as conditions improve; the producer moves toward the maximum. Movement toward the minimum and maximum are defined by two additional attributes that specify the rate of movement toward the minimum and maximum. Also, the need for movement toward the minimum and maximum is evaluated at a configured interval.

Flow Control options are described in following table:

Table 9-2 Flow Control Parameters

Attribute	Description
Flow Control Enabled	Determines whether a producer can be flow controlled by the JMS server.
Flow Maximum	The maximum number of messages per second for a producer that is experiencing a threshold condition. If a producer is not currently limiting its flow when a threshold condition is reached, the initial flow limit for that producer is set to Flow Maximum. If a producer is already limiting its flow when a threshold condition is reached (the flow limit is less than Flow Maximum), then the producer will continue at its current flow limit until the next time the flow is evaluated. Once a threshold condition has subsided, the producer is not permitted to ignore its flow limit. If its flow limit is less than the Flow Maximum, then the producer must gradually increase its flow to the Flow Maximum each time the flow is evaluated. When the producer finally reaches the Flow Maximum, it can then ignore its flow limit and send without limiting its flow.

Table 9-2 Flow Control Parameters

Attribute	Description
Flow Minimum	The minimum number of messages per second for a producer that is experiencing a threshold condition. This is the lower boundary of a producer's flow limit. That is, WebLogic JMS will not further slow down a producer whose message flow limit is at its Flow Minimum.
Flow Interval	An adjustment period of time, defined in seconds, when a producer adjusts its flow from the Flow Maximum number of messages to the Flow Minimum amount, or vice versa.
Flow Steps	The number of steps used when a producer is adjusting its flow from the Flow Minimum amount of messages to the Flow Maximum amount, or vice versa. Specifically, the Flow Interval adjustment period is divided into the number of Flow Steps (for example, 60 seconds divided by 6 steps is 10 seconds per step). Also, the movement (that is, the rate of adjustment) is calculated by dividing the difference between the Flow Maximum and the Flow Minimum into steps. At each Flow Step, the flow is adjusted upward or downward, as necessary, based on the current conditions, as follows: • The downward movement (the decay) is geometric over the specified period of time (Flow Interval) and according to the specified number of Flow Steps. (For example, 100, 50, 25, 12.5). • The movement upward is linear. The difference is simply divided by the number of Flow Steps.

For more information about the flow control fields, and the valid and default values for them, see "JMS Connection Factory: Configuration: Flow Control" in the *Administration Console Online Help*.

Flow Control Thresholds

The attributes used for configuring bytes/messages thresholds are defined as part of the JMS server and/or its destination. Table 9-3 defines how the upper and lower thresholds start and stop flow control on a JMS server and/or JMS destination.

Table 9-3 Flow Control Threshold Parameters

Attribute	Description
Bytes/Messages Threshold High	When the number of bytes/messages exceeds this threshold, the JMS server/destination becomes armed and instructs producers to limit their message flow.
Bytes/Messages Threshold Low	When the number of bytes/messages falls below this threshold, the JMS server/destination becomes unarmed and instructs producers to begin increasing their message flow. Flow control is still in effect for producers that are below their message flow maximum. Producers can move their rate upward until they reach their flow maximum, at which point they are no longer flow controlled.

For detailed information about other JMS server and destination threshold and quota fields, and the valid and default values for them, see the following pages in the *Administration Console Online Help*:

- JMS Server: Configuration: Thresholds and Quotas
- JMS Queue: Configuration: Thresholds and Quotas
- JMS Topic: Configuration: Thresholds and Quotas

Handling Expired Messages

The following sections describe two message expiration features, the message Expiration Policy and the Active Expiration of message, which provide more control over how the system searches for expired messages and how it handles them when they are encountered.

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 defined “error” destination.

- “Defining a Message Expiration Policy” on page 9-11
- “Enabling Active Message Expiration” on page 9-16

Defining a Message Expiration Policy

Use the message Expiration Policy feature to define an alternate action to take when messages expire. Using the Expiration Policy attribute on the Destinations node, an expiration policy can be set on a per destination basis. The Expiration Policy attribute defines the action that a destination should take when an expired message is encountered: discard the message, discard the message and log its removal, or redirect the message to an error destination.

Also, if you use JMS templates to configure multiple destinations, you can use the Expiration Policy field to quickly configure an expiration policy on all your destinations. To override a template’s expiration policy for specific destinations, you can modify the expiration policy on any destination.

For instructions on configuring the Expiration Policy, click one of the following links:

- “Configuring an Expiration Policy on Topics” on page 9-11
- “Configuring an Expiration Policy on Queues” on page 9-12
- “Configuring an Expiration Policy on Templates” on page 9-13
- “Defining an Expiration Logging Policy” on page 9-14

Configuring an Expiration Policy on Topics

Follow these directions if you are configuring an expiration policy on topics without using a JMS template. Expiration policies that are set on specific topics will override the settings defined on a JMS template.

1. Follow the directions for navigating to the JMS Topic: Configuration: Delivery Failure page in "Configure message delivery failure options" in the *Administration Console Online Help*.
2. From the Expiration Policy list box, select an expiration policy option.
   - Discard — Expired messages are removed from the system. The removal is not logged and the message is not redirected to another location.
   - Log — Removes expired messages and writes an entry to the server log file indicating that the messages were removed from the system. You define the actual information that will be logged in the Expiration Logging Policy field in next step.
   - Redirect — Moves expired messages from their current location into the Error Destination defined for the topic.

   For more information about the Expiration Policy options for a topic, see "JMS Topic: Configuration: Delivery Failure" in the *Administration Console Online Help*.
3. If you selected the Log expiration policy in previous step, use the Expiration Logging Policy field to define what information about the message is logged.

   For more information about valid Expiration Logging Policy values, see "Defining an Expiration Logging Policy" on page 9-14.
4. Click Save.

Configuring an Expiration Policy on Queues

Follow these directions if you are configuring an expiration policy on queues without using a JMS template. Expiration policies that are set on specific queues will override the settings defined on a JMS template.

1. Follow the directions for navigating to the JMS Queue: Configuration: Delivery Failure page in "Configure message delivery failure options" in the *Administration Console Online Help*.
2. From the Expiration Policy list box, select an expiration policy option.
   - Discard — Expired messages are removed from the system. The removal is not logged and the message is not redirected to another location.
   - Log — Removes expired messages from the queue and writes an entry to the server log file indicating that the messages were removed from the system. You define the actual information that will be logged in the Expiration Logging Policy field described in the next step.
   - Redirect — Moves expired messages from the queue and into the Error Destination defined for the queue.

For more information about the Expiration Policy options for a queue, see "Configure queue message delivery failure options" in the *Administration Console Online Help*.

3. If you selected the Log expiration policy in the previous step, use the Expiration Logging Policy field to define what information about the message is logged.

   For more information about valid Expiration Logging Policy values, see "Defining an Expiration Logging Policy" on page 9-14.

4. Click Save

5. Repeat steps 3–7 to configure an expiration policy for additional queues.

Configuring an Expiration Policy on Templates

Since JMS templates provide an efficient way to define multiple destinations (topics or queues) with similar attribute settings, you can configure a message expiration policy on an existing template (or templates) for your destinations.

1. Follow the directions for navigating to the JMS Template: Configuration: Delivery Failure page in "Configure message delivery failure options" in the *Administration Console Online Help*.

2. In the Expiration Policy list box, select an expiration policy option.

   – Discard — Expired messages are removed from the messaging system. The removal is not logged and the message is not redirected to another location.

   – Log — Removes expired messages and writes an entry to the server log file indicating that the messages were removed from the system. The actual information that is logged is defined by the Expiration Logging Policy field described in the next step.

   – Redirect — Moves expired messages from their current location into the Error Destination defined for the destination.

   For more information about the Expiration Policy options for a template, see "JMS Template: Configuration: Delivery Failure" in the *Administration Console Online Help*.

3. If you selected the Log expiration policy in Step 4, use the Expiration Logging Policy field to define what information about the message is logged.

   For more information about valid Expiration Logging Policy values, see "Defining an Expiration Logging Policy" on page 9-14.

4. Click Save.

5. Repeat steps 2–6 to configure an expiration policy for additional JMS templates.

Defining an Expiration Logging Policy

Note: The Expiration Logging Policy parameter has been deprecated in this release of WebLogic Server. In its place, BEA recommends using the Message Life Cycle Logging feature, which provide a more comprehensive view of the basic events that JMS messages will traverse through once they are accepted by a JMS server, including detailed message expiration data. For more information about message life cycle logging options, see "Message Life Cycle Logging" on page 8-7.

When the Expiration Policy is set to Log, the Expiration Logging Policy defines what information about the message is logged. Valid values for Expiration Logging Policy properties include`%header%`, `%properties%`, JMS header properties as defined in the JMS specification, the WebLogic JMS-specific extended header fields `JMSDeliveryTime` and `JMSRedeliveryLimit`, and any user-defined property. Each property must be separated by a comma.

The `%header%` value indicates that all header fields should be logged. The `%properties%` value indicates that all user properties should be logged. Neither values are case sensitive. However, the enumeration of individual JMS header fields and user properties are case sensitive.

For example, you could specify one of the following values:

- `JMSPriority, Name, Address, City, State, Zip`
- `%header%, Name, Address, City, State, Zip`
- `JMSCorrelationID, %properties%`

The `JMSMessageID` field is always logged and cannot be turned off. Therefore, if the Expiration Policy is not defined (that is, none) or is defined as an empty string, then the output to the log file contains only the `JMSMessageID` of the message.

Expiration Log Output Format

When an expired message is logged, the text portion of the message (not including timestamps, severity, thread information, security identity, etc.) conforms to the following format:

```
<ExpiredJMSMessage JMSMessageId='$MESSAGEID' >
    <HeaderFields Field1='Value1' [Field2='Value2'] … ] />
    <UserProperties Property1='Value1' [Property='Value2'] … ] />
</ExpiredJMSMessage>
```

where `$MESSAGEID` is the exact string returned by `Message.getJMSMessageID()`.

For example:

```
<ExpiredJMSMessage JMSMessageID='ID:P<851839.1022176920343.0' >
    <HeaderFields JMSPriority='7' JMSRedelivered='false' />
    <UserProperties Make='Honda' Model='Civic' Color='White'
     Weight='2680' />
</ExpiredJMSMessage>
```

If no header fields are displayed, the line for header fields is not be displayed. If no user properties are displayed, that line is not be displayed. If there are no header fields and no properties, the closing `</ExpiredJMSMessage>` tag is not necessary as the opening tag can be terminated with a closing bracket (`/>`).

For example:

```
<ExpiredJMSMessage JMSMessageID='ID:N<223476.1022177121567.1' />
```

All values are delimited with double quotes. All string values are limited to 32 characters in length. Requested fields and/or properties that do not exist are not displayed. Requested fields and/or properties that exist but have no value (a null value) are displayed as null (without single quotes). Requested fields and/or properties that are empty strings are displayed as a pair of single quotes with no space between them.

For example:

```
<ExpiredJMSMessage JMSMessageID='ID:N<851839.1022176920344.0' >
    <UserProperties  First='Any string longer than 32 char ...'
     Second=null Third='' />
</ExpiredJMSMessage>
```

Enabling Active Message Expiration

Use the Active Expiration feature to define the timeliness in which expired messages are removed from the destination to which they were sent or published. Messages are not necessarily removed from the system at their expiration time, but they are removed within a user-defined number of seconds. The smaller the window, the closer the message removal is to the actual expiration time.

Configuring a JMS Server to Actively Scan Destinations for Expired Messages

Follow these directions to define how often a JMS server will actively scan its destinations for expired messages. The default value is 30 seconds, which means the JMS server waits 30 seconds between each scan interval.

1. Follow the directions for navigating to the JMS Server: Configuration: General page of the Administration Console in "Configure general JMS server properties" in the *Administration Console Online Help*.
2. In the Scan Expiration Interval field, enter the amount of time, in seconds, that you want the JMS server to pause between its cycles of scanning its destinations for expired messages to process.

   To disable active scanning, enter a value of 0 seconds. Expired messages are passively removed from the system as they are discovered.

   For more information about the Expiration Scan Interval attribute, see "JMS Server: Configuration: General" in the *Administration Console Online Help*.
3. Click Save.

Index