Oracle® WebLogic Server

Configuring and Managing WebLogic JMS

10*g* Release 3 (10.3)

July 2008

ORACLE

Oracle WebLogic Server Configuring and Managing WebLogic JMS, 10*g* Release 3 (10.3)

Contents

1. Introduction and Roadmap

2. Understanding JMS Resource Configuration

3. Configuring Basic JMS System Resources

4. Configuring Advanced JMS System Resources

5. Configuring JMS Application Modules for Deployment

8. Troubleshooting WebLogic JMS

CHAPTER 1

Introduction and Roadmap

The following sections describe the contents and organization of this guide—*Configuring and Managing WebLogic JMS*.

Document Scope and Audience

This document is a resource for system administrators who configure, manage, and monitor WebLogic JMS resources, including JMS servers, stand-alone destinations (queues and topics), distributed destinations, and connection factories.

The document is relevant to production phase administration, monitoring, and performance tuning. It 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," describes the organization and scope of this guide.
- Chapter 2, "Understanding JMS Resource Configuration," is an overview of WebLogic JMS architecture and features.
- Chapter 3, "Configuring Basic 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 Advanced JMS System 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 stand-alone 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 Module Resources," explains how to use the WebLogic Scripting Tool to create and manage JMS resources programmatically.
- 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 information on developing, deploying, and monitoring WebLogic Server applications:

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

- Understanding WebLogic Server Clustering in *Using Clusters* 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 Server Environments* describes the benefits and use of the system-wide WebLogic Persistent Store.
- *Configuring and Managing WebLogic Store-and-Forward* describes the benefits and use 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.
- *Performance and Tuning* contains information on monitoring and improving the performance of WebLogic Server applications, including information on how to get the most out of your JMS applications by using the administrative performance tuning features available with WebLogic JMS.

JMS Samples and Tutorials for the JMS Administrator

In addition to this document, Oracle provides JMS code samples and tutorials that document JMS configuration, API use, and key JMS development tasks. Oracle 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 Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application enables patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights Oracle-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.

JMS Examples in the WebLogic Server Distribution

This release of WebLogic Server 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 Start menu.

New and Changed JMS Features In This Release

For a comprehensive listing of the new WebLogic Server features introduced in this release, see What's New in WebLogic Server in *Release Notes*.

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 secure Java EE 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 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.

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 *Using WebLogic JTA*.
- File or database persistent message storage (both fully XA transaction capable). See Using the WebLogic Persistent Store in *Configuring Server Environments*.

- Message Store-and-Forward (SAF) is clusterable and improves reliability by locally storing messages sent to unavailable remote destinations. See Understanding the Store-and-Forward Service in *Configuring and Managing WebLogic Store-and-Forward*.
- If a server or network failure occurs, JMS producer and consumer objects will attempt to transparently failover to another server instance, if one is available. See Automatic JMS Client Failover in *Programming WebLogic JMS*.
- Supports connection clustering using connection factories targeted on multiple WebLogic Servers, as described in "Configuring WebLogic JMS Clustering" on page 4-1.
- System-assisted configuration of Uniform Distributed Destinations that provide high availability, load balancing, and failover support in a cluster, as described in Using Distributed Destinations in *Programming WebLogic JMS*.
- Automatic whole server migration provides improved cluster reliability and server migration WebLogic Server now supports automatic and manual migration of a clustered server instance and all the services it hosts from one machine to another, as described in "Configuring WebLogic JMS Clustering" on page 4-1.
- Redirects failed or expired messages to error destinations, as described in Managing Rolled Back, Recovered, Redelivered, or Expired Messages in *Programming WebLogic JMS*.
- Supports the JMS Delivery Count message property (`JMSXDeliveryCount`), which specifies the number of message delivery attempts, where the first attempt is 1, the second is 2, and so on. WebLogic Server makes a best effort to persist the delivery count, so that the delivery count does not reset back to one after a server reboot. See Message in *Programming WebLogic JMS*.
- *Provides three levels of load balancing*: network-level, JMS connections, and distributed destinations.

Enterprise-level Features

- WebLogic Server fully supports the JMS 1.1 Specification, is fully compliant with the Java EE 5.0 specification, and provides numerous WebLogic JMS Extensions that go beyond the standard JMS APIs.
- Provides robust message and destination management capabilities:
  - Administrators can manipulate most messages in a running JMS Server, using either the Administration Console or runtime APIs. See "Managing JMS Messages" on page 7-5.

  - Administrators can 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, using either the Administration Console or runtime APIs. See "Controlling Message Operations on Destinations" on page 8-15.
  - Message-Driven EJBs (MDBs) also supply message pause and resume functionality, and can even automatically temporarily pause during error conditions. See Message-Driven EJBs in *Programming WebLogic Enterprise JavaBeans*.

- Modular deployment of JMS resources, which are defined by an XML so that you can migrate your application and the required JMS configuration from environment to environment without opening an enterprise application file, and without extensive manual JMS reconfiguration. See "Overview of JMS Modules" on page 2-7.
- JMS message producers can group ordered messages into a single unit-of-order, which guarantees that all such messages are processed serially in the order in which they were created. See Using Message Unit-of-Order in *Programming WebLogic JMS*.
- To provide an even more restricted notion of a group than the Message Unit-of-Order feature, the Message Unit-of-Work (UOW) feature allows JMS producers to identify certain messages as components of a UOW message group, and allows a JMS consumer to process them as such. For example, a JMS producer can designate a set of messages that need to be delivered to a single client without interruption, so that the messages can be processed as a unit. See Using Unit-of-Work Message Groups in *Programming WebLogic JMS*.
- 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. See "Message Life Cycle Logging" on page 8-7.
- Timer services available for scheduled message delivery, as described in Setting Message Delivery Times in *Programming WebLogic JMS*.
- Flexible expired message policies to handle expired messages, as described in Handling Expired Messages in *Performance and Tuning*.
- Supports messages containing *XML* (Extensible Markup Language). See Defining XML Message Selectors Using the XML Selector Method in *Programming WebLogic JMS*.
- Thin application client *.JAR that provides full WebLogic Server Java EE functionality*, including JMS, yet greatly reduces the client-side WebLogic footprint. See WebLogic JMS Thin Client in *Programming Stand Alone Clients*.

- The JMS SAF Client enables standalone JMS clients to reliably send messages to server-side JMS destinations, even when the JMS client cannot temporarily reach a destination (for example, due to a network connection failure). While disconnected from the server, messages sent by the JMS SAF client are stored locally on the client and are forwarded to server-side JMS destinations when the client reconnects. See Reliably Sending Messages Using the JMS SAF 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. See Enhanced Java EE Support for Using WebLogic JMS With EJBs and Servlets in *Programming WebLogic JMS*.

Performance

WebLogic JMS features enterprise-class performance features, such as automatic message paging, message compression, and DOM support for XML messages:

- WebLogic Server uses highly optimized disk access algorithms and other internal enhancements to provide a unified messaging kernel that improves both JMS-based and Web Services messaging performance. See Using the WebLogic Persistent Store in *Configuring Server Environments*.
- You may greatly improve the performance of typical non-persistent messaging with One-Way Message Sends. When configured on a connection factory, associated producers can send messages without internally waiting for a response from the target destination's host JMS server. You can choose to allow queue senders and topic publishers to do one-way sends, or to limit this capability to topic publishers only. You can also specify a "One-Way Window Size" to determine when a two-way message is required to regulate the producer before it can continue making additional one-way sends. See Configure connection factory flow control in the *Administration Console Online Help*.
- Message paging automatically kicks in during peak load periods to free up virtual memory. See Paging Out Messages To Free Up Memory in *Performance and Tuning*.
- Administrators can enable the compression of messages that exceed a specified threshold size to improve the performance of sending messages travelling across JVM boundaries using either the Administration Console or runtime APIs. See Compressing Messages in *Performance and Tuning*.
- Synchronous consumers can also use the same efficient behavior as asynchronous consumers by enabling the Prefetch Mode for Synchronous Consumers option on the consumer's JMS connection factory, using either the Administration Console or runtime

APIs. See Using the Prefetch Mode to Create a Synchronous Message Pipeline in *Programming WebLogic JMS*

- Supplies a wide variety of performance tuning options for JMS messages. See Tuning WebLogic JMS in *Performance and Tuning*.
- Supports MDB transaction batching by processing multiple messages in a single transaction. See Transaction Batching of MDBs in *Programming WebLogic Enterprise JavaBeans*.
- JMS SAF provides better performance than the WebLogic Messaging Bridge across clusters. See Tuning WebLogic JMS Store-and-Forward in *Performance and Tuning*.
- DOM (Document Object Model) support for sending XML messages greatly improves performance for implementations that already use a DOM, since those applications do not have to flatten the DOM before sending XML messages. See Sending XML Messages in *Programming WebLogic JMS*.
- Message flow control during peak load periods, including blocking overactive senders, as described in Controlling the Flow of Messages on JMS Servers and Destinations and Defining Quota in *Performance and Tuning*.
- The automatic pooling of connections and other objects by the JMS wrappers via JMS resource-reference pooling. See Enhanced Java EE Support for Using WebLogic JMS With EJBs and Servlets in *Programming WebLogic JMS*.
- Multicasting of messages for simultaneous delivery to many clients using IP multicast, as described in Using Multicasting with WebLogic JMS 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, Enterprise Java Beans (including MDBs), 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.

- Supports the WebLogic Scripting Tool (WLST) to initiate, manage, and persist configuration changes interactively or by using an executable script. See Chapter 6, "Using WLST to Manage JMS Servers and JMS System Module Resources."
- 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 *Using WebLogic JTA*.
- Leverages sophisticated security model built into WebLogic Server (policy engine), as described in the *Understanding WebLogic Security* and Resource Types You Can Secure with Policies in *Securing WebLogic Resources*.

Interoperability With Other Messaging Services

- Fully supports direct interoperability from WebLogic Server 8.1 through WebLogic Server 10.0. For example, a release 8.1 client can interoperate with a release 10.0 server and vice-versa.
- 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 the WebLogic Messaging Bridge*.
- Supports mapping of other JMS providers so their objects appear in the WebLogic JNDI tree as local JMS objects. Also references remote instances of WebLogic Server in another cluster or domain in the local JNDI tree. See "Foreign Server Configuration" on page 3-18.
- Uses MDBs to transactionally receive messages from multiple JMS providers. See Message-Driven EJBs in *Programming WebLogic Enterprise JavaBeans.*
- Reliable Web Services integration with JMS as a transport, as described in Using Web Services Reliable Messaging in *Programming Advanced Features of WebLogic Web Services Using JAX-RPC.*
- Automatic transaction enlistment of non-WebLogic JMS client resources in server-side applications via JMS resource-reference pooling. See Enhanced Java EE Support for Using WebLogic JMS With EJBs and Servlets in *Programming WebLogic JMS*.
- Integration with Oracle Tuxedo messaging provided by WebLogic Tuxedo Connector. See How to Configure the Oracle Tuxedo Queuing Bridge in the *Oracle WebLogic Tuxedo Connector Administration Guide*.

- 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). See WebLogic JMS C API in *Programming WebLogic JMS*.

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:

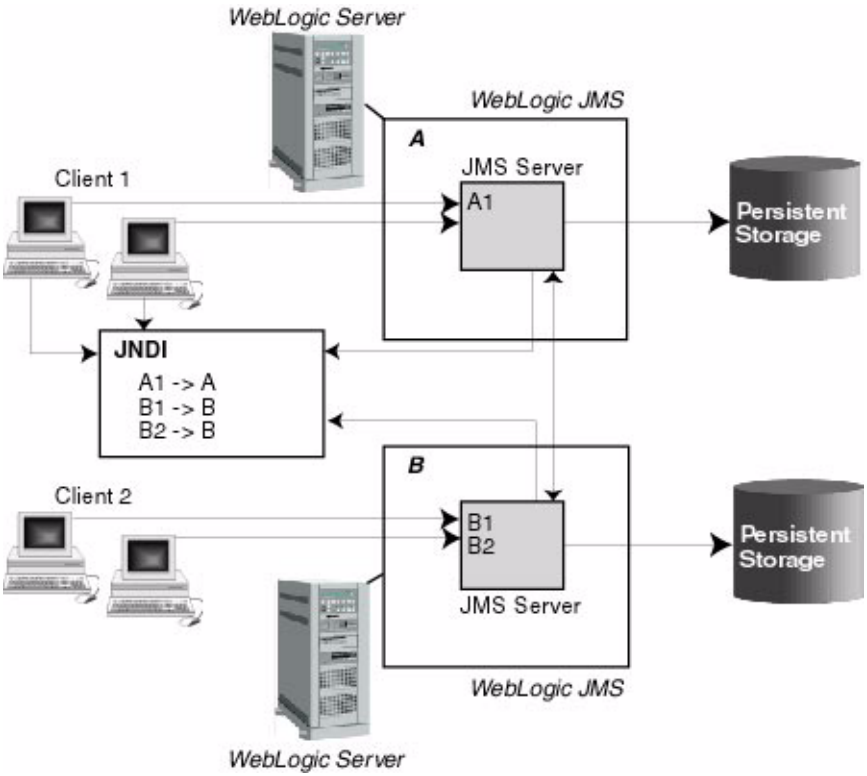
- Enables Java applications that share a messaging system to exchange messages
- Simplifies application development by providing a standard interface for creating, sending, and receiving messages

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 Architecture and Environment

The following figure illustrates the WebLogic JMS architecture.

Figure 2-1 WebLogic JMS Architecture



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

The major components of the WebLogic JMS architecture include:

- A JMS server is an environment-related configuration entity that acts as management container for JMS queue and topic resources defined within JMS modules that are targeted to specific that 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 configure one or more JMS servers per domain, and a JMS server can manage one or more JMS modules. For more information, see "Overview of JMS Servers" on page 2-5.

- JMS modules contain configuration resources, such as standalone queue and topic destinations, distributed destinations, and connection factories, and are defined by XML documents that conform to the `weblogic-jms.xsd` schema. For more information, see "What Are JMS Configuration Resources?" on page 2-5.
- 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: Environment-Related Resources versus Application-Related 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* information and *application-related* information. Examples of environment-related information 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 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.

Examples of application-related definitions that are independent of the domain environment are the various Java EE application components configurations, such as EAR, WAR, JAR, RAR files, and 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.

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 life cycle and tuning, any modification to these applications must be completed by the application development team.

What Are JMS Configuration Resources?

Beginning in WebLogic Server 9.0, JMS configuration resources, such as destinations and connections factories, are stored outside of the WebLogic domain as module descriptor files, which are defined by XML documents that conform to the `weblogic-jms.xsd` schema. JMS modules do not include JMS server definitions, which are stored in the WebLogic domain configuration file, as described in "Overview of JMS Servers" on page 2-5.

You create and manage JMS resources either as *system 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 Java EE modules and can be deployed either with a Java EE application (as a packaged resource) or as stand-alone modules that can be made globally available. See "Overview of JMS Modules" on page 2-7.

Overview of JMS Servers

JMS servers are environment-related configuration entities that act as management containers for destination 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.

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.

JMS Server Behavior in WebLogic Server 9.0 and Later

Beginning in WebLogic Server 9.0, JMS server behavior differs in certain respects from behavior in pre-9.0 releases:

- Because destination resources are encapsulated in JMS modules, they are not nested under JMS servers in the configuration file. However, a *sub-targeting* relationship between JMS servers and destinations is maintained because each standalone destination resource within a JMS module is always targeted to a single JMS server. This way, JMS servers continue to manage persistent messages, durable subscribers, message paging, and, optionally, quotas for their targeted destinations. Multiple JMS modules can be targeted to each JMS server in a domain.
- JMS servers support the default 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 for backward compatibility.)
- JMS servers support an improved message paging mechanism. For more information on message paging, see *Performance and Tuning*.
  - 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 3-5.

Overview of JMS Modules

JMS modules are application-related definitions that are independent of the domain environment. You create and manage JMS resources either as *system modules* or as *application modules*. JMS system modules are typically configured using the Administration Console or the WebLogic Scripting Tool (WLST), which adds a reference to the module in the domain's `config.xml` file. JMS application modules are a WebLogic-specific extension of Java EE modules and can be deployed either with a Java EE application (as a packaged resource) or as stand-alone modules that can be made globally available.

The main difference between system modules and application modules comes down to ownership. System modules are owned and modified by the WebLogic administrator and are available to all applications. Application modules are owned and modified by the WebLogic developers, who package the JMS resource modules with the application's EAR file.

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 stand-alone JMS module, and without extensive manual JMS reconfiguration.

These sections describe the different types of JMS module and the resources that they can contain:

JMS System Modules

WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JMS modules, and to configure the module's configuration resources, such as queues, and topics connection factories.

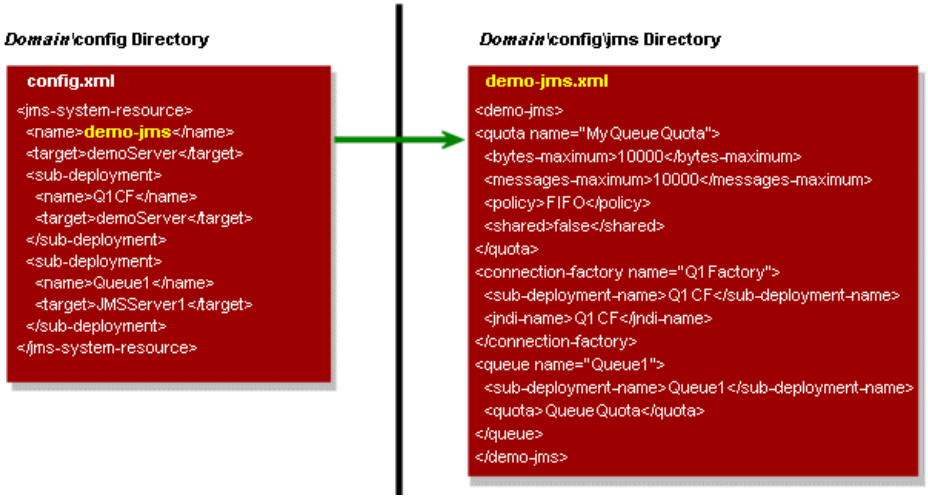
JMS modules that you configure this way are considered *system modules*. JMS system modules are owned by the Administrator, who can at any time add, modify, or delete resources. System modules are globally available for targeting to servers and clusters configured in the domain, and therefore are available to all applications deployed on the same targets and to client applications.

When you create a JMS system module 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 system module file and a list of target servers and clusters on which the module is deployed.

The JMS module conforms to the `weblogic-jms.xsd` schema, as described in "JMS Schema" on page 5-2. System modules are also accessible through WebLogic Management Extension (JMX) utilities, 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



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

JMS Application Modules

JMS configuration resources can also be managed as deployable application modules, similar to standard Java EE descriptor-based modules. JMS Application modules can be deployed either with a Java EE 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 developers typically create application 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, manage, and tune.

As discussed in "Domain Configuration: Environment-Related Resources versus Application-Related 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 configuring JMS application modules, see Chapter 5, "Configuring JMS Application Modules for Deployment."

Comparing JMS System Modules and Application Modules

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:

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 Java EE 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
Application	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 5-1 and Deploying Applications and Modules with weblogic.deployer in *Deploying Applications to WebLogic Server*.

Configurable JMS Resources in Modules

The following configuration resources are defined as part of a system module or an application module:

- Queue and topic destinations, as described in "Queue and Topic Destination Configuration" on page 3-13.
- Connection factories, as described in "Connection Factory Configuration" on page 3-11.
- Templates, as described in "JMS Template Configuration" on page 3-16.
- Destination keys, as described in "Destination Key Configuration" on page 3-17.
- Quota, as described in "Quota Configuration" on page 3-18.
- Distributed destinations, as described in "Configuring Distributed Destination Resources" on page 4-13.
- Foreign servers, as described in "Configuring Foreign Server Resources to Access Third-Party JMS Providers" on page 4-10.

- JMS store-and-forward (SAF) configuration items, as described in "JMS Store-and-Forward (SAF)" on page 2-12.

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

- JMS servers (required), as described in "Overview of JMS Servers" on page 2-5
- Store-and-Forward agents (optional), as described in "JMS Store-and-Forward (SAF)" on page 2-12.
- Path service (optional), as described in "Path Service" on page 2-13.
- 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 Basic JMS System Resources" on page 3-1.

JMS Schema

In support of the modular configuration model for JMS resources, Oracle provides a schema for WebLogic JMS objects: `weblogic-jms.xsd`. 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-jms.xsd` schema is available online at `http://www.bea.com/ns/weblogic/weblogic-jms/1.0/weblogic-jms.xsd`.

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.

A JMS interop module can also implement many of the WebLogic Server 9.0 or later features, such as message unit-of-order and destination quota. However, it *cannot* implement the following WebLogic Server 9.0 or later features:

- Uniform distributed destination resources
- JMS store-and forward resources

Caution: Use of any new features in the current release in a JMS interop module may possibly break compatibility with JMX clients prior to WebLogic Server 9.0.

Other Environment-Related System Resources for WebLogic JMS

These environment-related resources must be configured by the administrator as domain configuration resources in order to be accessible to JMS Servers and JMS modules.

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 Server Environments*.

JMS Store-and-Forward (SAF)

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 Configuring SAF for JMS Messages 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 4-8.

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 the WebLogic Messaging Bridge*.

CHAPTER 3

Configuring Basic JMS System Resources

The procedures in the following sections describe how to configure and manage basic JMS system resources, such as JMS servers and JMS system modules.

Methods for Configuring JMS System Resources

WebLogic Administrators can use these tools to create and deploy (target) system resources, such as JMS servers and JMS system modules.

- 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 3-5.
  - JMS system modules, as described in "JMS System Module Configuration" on page 3-7.
  - Store-and-Forward services for JMS, as described in Configuring SAF for JMS Messages in *Configuring and Managing WebLogic Store-and-Forward*.
  - Persistent stores, as described in Using the WebLogic Persistent Store in *Configuring Server Environments*.
- 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. See Chapter 6, "Using WLST to Manage JMS Servers and JMS System Module Resources."
- WebLogic Java Management Extensions (JMX) is the Java EE solution for monitoring and managing resources on a network. See Overview of WebLogic Server Subsystem MBeans in *Developing Custom Management Utilities with JMX*.
- 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 to Manage Applications in *Programming WebLogic JMS* or the JMSModuleHelper Class Javadoc.

Note: For information on configuring and deploying JMS application modules in an enterprise application, see Chapter 5, "Configuring JMS Application Modules for Deployment."

Main Steps for Configuring Basic JMS System Resources

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

WebLogic JMS provides default values for some configuration options; you must provide values for all others. Once WebLogic JMS is configured, applications can send and receive messages using the JMS API. For information on tuning the default configuration parameters, see *Performance and Tuning* or `JMSBean` in the *WebLogic Server MBean Reference*.

1. If you require persistent messaging, use one of the following storage options:
   - To store 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. See Creating a Custom (User-Defined) File Store in the *Configuring Server Environments*.
   - To store persistent messages in a JDBC-accessible database, you must create a JDBC store. See Creating a JDBC Store in *Configuring Server Environments*.
2. Configure a JMS server to manage the messages that arrive on the queue and topic destinations in a JMS system module. See "Overview of JMS Servers" on page 2-5.
3. Configure a JMS system module to contain your destinations, as well as other resources, such as quotas, templates, destination keys, distributed destinations, and connection factories. See "JMS System Modules" on page 2-7.
4. Before creating any queues or topics 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 for your destinations. Destinations can be assigned their own quotas; multiple destinations can share a quota; or destinations can share the JMS server's quota. See "Quota Configuration" on page 3-18.
   - Create JMS templates, which allow you to define multiple destinations with similar option settings. See "JMS Template Configuration" on page 3-16.
   - Configure destination keys to create custom sort orders of messages as they arrive on a destination. See "Destination Key Configuration" on page 3-17.

   Once these resources are configured, you can select them when you configure your queue or topic resources.
5. Configure a queue and/or topic destination in your system module:
   - Configure a standalone topic for the delivery of messages to multiple recipients (publish/subscribe). See "Queue and Topic Destination Configuration" on page 3-13.
   - Configure a standalone queue for the delivery of messages to exactly one recipient (point-to-point). See "Queue and Topic Destination Configuration" on page 3-13.

6. 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 3-11. For more information on configuring a Connection Factory, see "Connection Factory Configuration Parameters" on page 3-12.

WebLogic JMS provides default values for some configuration options; you must provide values for all others. Once WebLogic JMS is configured, applications can send and receive messages using the JMS API.

Advanced Resources in JMS System Modules

Beyond basic JMS resource configuration, you can add these advanced resources to a JMS system module:

- Create a Uniform Distributed Destination resource to configure a set of queues or topics that distributed across the cluster, with each member belonging to a separate JMS server in the cluster. See "Configuring Distributed Destination Resources" on page 4-13.
- Create a JMS Store-and-Forward resource to reliably forward messages to remote destinations, even when 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 resource to reference third-party JMS providers within a local WebLogic Server JNDI tree. See "Configuring Foreign Server Resources to Access Third-Party JMS Providers" on page 4-10.

JMS Configuration Naming Requirements

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 configuration objects, including 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 (for example, queues, topics, and 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.

Note: A sample `examplesJMSServer` configuration is provided with the product in the Examples Server. For more information about developing basic WebLogic JMS applications, refer to Developing a Basic JMS Application in *Programming WebLogic JMS*.

JMS Server Configuration Parameters

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

You can configure the following parameters for JMS servers:

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

  For more information about configuring messages and bytes quota for JMS servers and destinations, see *Performance and Tuning*.
- Message logging parameters for a JMS server's log file, which contains the basic events that a JMS message traverses, 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 and 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` and `JMSServerRuntimeMBean` in the *WebLogic Server MBean Reference*.

JMS Server Targeting

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 — 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.
- Migratable Target — Migratable targets define a set of WebLogic Server instances in a cluster that can potentially host an *exactly-once* service, such as a JMS server. When a migratable target server boots, the JMS server boots as well on the specified *user-preferred* server in the cluster. However, a JMS server and all of its destinations can be migrated 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 services, see "Migration of JMS-related Services" on page 4-6.

For instructions on specifying JMS server targets using the Administration Console, see Change JMS server targets in the *Administration Console Online Help*.

JMS Server Monitoring Parameters

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.

Session Pools and Connection Consumers

Note: Session pool and connection consumer configuration objects were deprecated in WebLogic Server 9.0. 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. 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. 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 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*.

You define the following “basic” configuration resources as part of a JMS system module:

- Queue and topic destinations, as described in “Queue and Topic Destination Configuration” on page 3-13.
- Connection factories, as described in “Connection Factory Configuration” on page 3-11.
- Templates, as described in “JMS Template Configuration” on page 3-16.
- Destination keys, as described in “Destination Key Configuration” on page 3-17.

- Quota, as described in "Quota Configuration" on page 3-18.

You can also define the following "advanced" clustering configuration resources as part of a JMS system module:

- Foreign servers, as described in "Configuring Foreign Server Resources to Access Third-Party JMS Providers" on page 4-10.
- Distributed destinations, as described in "Configuring Distributed Destination Resources" on page 4-13.
- JMS store-and-forward configurations, as described in Configuring SAF for JMS Messages in *Configuring and Managing WebLogic Store-and-Forward*.

A sample `examples-jms` module 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*.

For information on alternative methods for configuring JMS system modules, such as using the WebLogic Scripting Tool (WLRT), see "Methods for Configuring JMS System Resources" on page 3-2.

JMS System Module and Resource Subdeployment Targeting

JMS system modules must be targeted to one or more WebLogic Server instances or to a cluster. Targetable JMS 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.

Default Targeting

When using the Administration Console to configure resources in a JMS system module, you can choose whether to simply accept the parent module's default targets or to proceed to an advanced targeting page where you can use the subdeployment mechanism for targeting the resource. However, standalone queue and topic resource types, cannot use default targets and must be targeted to a subdeployment that is targeted to a single JMS server.

When you select the default targeting mechanism, it's target status will be reflected by the Default Targeting Enabled check box on the resource type's Configuration: General page on the Administration Console.

For more information on configuring JMS system resources, see Configure resources for JMS system modules in the *Administration Console Online Help*.

Subdeployment Targeting

When targeting standalone queue and topic resources, or when bypassing the default targeting mechanism for other resource types, you must use subdeployment targets. A subdeployment is a mechanism by which targetable system module resources (such as standalone destinations, distributed destinations, 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, which would be the case, for example, if a connection factory is targeted to a cluster that is hosting JMS servers in a domain. UDDs, however, 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
Uniform distributed queue	JMS server(s) | server instance(s) | cluster
Uniform 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

Note: Connection factory, uniform distributed destination, foreign server, and SAF imported destination resources can also be configured to default to their parent module's targets, as explained in "Default Targeting" on page 3-8.

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/91">
  <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>
```

To help manage your subdeployments for a JMS system module, the Administration Console provides subdeployment management pages. For more information, see Configure subdeployments in JMS system modules in the *Administration Console Online Help*.

For information about deploying stand-alone JMS modules, see Deploying JDBC, JMS, and WLDF Application Modules in *Deploying Applications to WebLogic Server*.

Connection Factory Configuration

Connection factories are resources 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 3-11.

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. And, 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 3-4. 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 Server instances. For more information on configuring JMS clustering, see "Configuring WebLogic JMS Clustering" on page 4-1.

Using a Default Connection Factory

WebLogic Server 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*.

Also, using default connection factories means 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: Configuration: Services in the *Administration Console Online Help*.

Connection Factory Configuration Parameters

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*.

You can modify the following parameters for connection factories:

- 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.

  Note: When selecting the "XA Connection Factory Enabled" option to enable JTA transactions with JDBC stores, you must verify that the configured JDBC data source uses a non-XA JDBC driver. This limitation does not remove the XA capabilities of layered subsystems that use JDBC stores. For example, WebLogic JMS is fully XA-capable regardless of whether it uses a file store or any JDBC store.
- 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*.

Connection Factory Targeting

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 "JMS System Module and Resource Subdeployment Targeting" on page 3-8.

Queue and Topic Destination Configuration

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 — Assign quotas to destinations; multiple destinations can share a quota; or destinations can share the JMS server's quota. See *Performance and Tuning*.

– JMS Template — Define multiple destinations with similar option settings. You also need a JMS template to create temporary queues. See "JMS Template Configuration" on page 3-16.

– Destination Key — Create custom sort orders of messages as they arrive on a destination. See "Destination Key Configuration" on page 3-17.

Queue and Topic Configuration Parameters

A JMS queue defines a *point-to-point* destination type for a JMS server. A message delivered to a queue is distributed to a single consumer. 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 is distributed to all consumers that are subscribed to that topic.

The WebLogic Server Administration Console enables you to configure, modify, target, and delete queue and topic resources in a system module. For a road map of queue and topic tasks, see Configure queues and Configure topics in the *Administration Console Online Help*. Within each JMS module, queue and topic resource names must be unique. And, all queue and topic JNDI names in any JMS module must be unique across an entire WebLogic domain, as defined in "JMS Configuration Naming Requirements" on page 3-4.

You can configure the following parameters for a queue and/or a topic:

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

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

- Threshold and quota parameters, which define the upper and lower message and byte threshold and maximum quota options for the destination. See "Quota Configuration" on page 3-18.

- Message logging parameters, such as message type and user properties, and logging message life cycle information into a JMS log file.

  See "Message Life Cycle Logging" on page 8-7. Pause and resume controls for message production, message insertion (in-flight messages), and message consumption operations on a destination. See "Controlling Message Operations on Destinations" on page 8-15.

- Message delivery override parameters, such as message priority and time-to-deliver values, which 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.
- For topics only, multicast parameters, including a multicast address, time-to-live (TTL), and port.

Some 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 options, see `QueueBean` and `TopicBean` in the *WebLogic Server MBean Reference*.

Creating 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*.

Creating Distributed Destinations

A distributed destination resource is a group of destinations (queues or topics) that are accessible as a single, logical unit to a client (for example, a distributed topic has its own JNDI name). The members of the set are typically distributed across multiple servers within a cluster, with each member belonging to a separate JMS server. See "Distributed Destination Configuration" on page 3-18.

Queue and Topic Targeting

Stand-alone 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 "JMS System Module and Resource Subdeployment Targeting" on page 3-8.

Destination Monitoring and Management Parameters

You can monitor run-time statistics for queues and topics in system modules, as well as manage the messages on queues and durable subscribers on topics.

- For information on using the Administration Console to monitor queues, see Monitoring queues in JMS system modules in the *Administration Console Online Help*.
- For information on managing messages on queues, as described in "Managing JMS Messages" on page 7-5.
- For more information on using the Administration Console to monitor topics, see Monitor topics in JMS system modules in the *Administration Console Online Help*.
- For information on managing durable subscriber on topics, as described in "Managing JMS Messages" on page 7-5.

JMS Template Configuration

A JMS template is an efficient means of defining multiple destinations with similar option settings:

- 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.

JMS Template Configuration Parameters

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*.

The configurable options for a JMS template are the same as those configured for a destination. See "Queue and Topic Configuration Parameters" on page 3-14.

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. See "JMS Configuration Naming Requirements" on page 3-4.
- The JNDI Name, Enable Store, and Template options are not defined for JMS templates.
- You can configure 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.

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.

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 Configuration

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).

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 Configuration

A quota resource 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.

See *Performance and Tuning*.

Foreign Server Configuration

A foreign server resource enables you to reference 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.

See "Configuring Foreign Server Resources to Access Third-Party JMS Providers" on page 4-10.

Distributed Destination Configuration

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

See "Configuring Distributed Destination Resources" on page 4-13.

JMS Store-and-Forward (SAF) Configuration

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.

See Configuring SAF for JMS Messages in *Configuring and Managing WebLogic Store-and-Forward*.

CHAPTER 4

Configuring Advanced JMS System Resources

These sections provide information on configuring advanced WebLogic JMS resources, such as a distributed destination 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 Destination Resources" on page 4-13.

  – *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 Whole Server Migration in *Using 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 Configuration" on page 3-11.

- *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 Whole Server Migration in *Using 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 manual migrations as well as automatic migrations in response to a WebLogic Server failure. For more information, see "Migration of JMS-related Services" on page 4-6.
- *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 Clusters*.

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

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 "Connection Factory Configuration Parameters" on page 3-12.

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 3-5.

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 3-4.

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 Destination Resources" on page 4-13.

JMS Services 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, JMS services are also part of the singleton service migration

framework. This allows an administrator, for example, to migrate a JMS server and all of its destinations to migrate to another WebLogic Server within a cluster in response to a server failure or for scheduled maintenance. This includes both scheduled migrations as well as automatic migrations. For more information on JMS service migration, see "Migration of JMS-related Services" on page 4-6.

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 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 Configuration" on page 3-11.
3. Optionally, identify migratable server targets for JMS services using the *Administration Console*. For example, 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 "Migration of JMS-related Services" on page 4-6. For more information about JMS server configuration attributes, see "JMS Server Configuration" on page 3-5.

   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 virtual distributed destination set, as discussed in "Distributed Destination Within a Cluster" on page 4-4.

What About Failover?

If a server or network failure occurs, JMS message producer and consumer objects will attempt to transparently failover to another server instance, if one is available. In WebLogic Server release 9.1 or later, WebLogic JMS message producers automatically attempt to reconnect to an available server instance without any manual configuration or changes to existing client code. In

WebLogic Server release 9.2 or later, you can use the Administration Console or WebLogic JMS APIs to configure WebLogic JMS message consumers to attempt to automatically reconnect to an available server instance. See Automatic JMS Client Failover in *Programming WebLogic JMS*.

Note: For WebLogic Server 9.0 or earlier JMS client applications, refer to Programming Considerations for WebLogic Server 9.0 or Earlier Failures in *Programming WebLogic JMS*.

In addition, implementing the automatic service migration feature ensures that exactly-once services, like JMS, do not introduce a single point of failure for dependent applications in the cluster. See "Migration of JMS-related Services" on page 4-6. WebLogic Server also supports data 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. See Whole Server Migration in *Using 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. See "Distributed Destination Within a Cluster" on page 4-4.

Oracle also recommends implementing high-availability clustering software such as VERITAS Cluster Server, which provides an integrated, out-of-the-box solution for WebLogic Server-based applications. Another recommended high-availability software solution is IBM HACMP or the equivalent.

Migration of JMS-related Services

JMS-related services are singleton services, and, therefore, are not active on all server instances in a cluster. Instead, they are pinned to a single server in the cluster to preserve data consistency. To ensure that singleton JMS services do not introduce a single point of failure for dependent applications in the cluster, WebLogic Server can be configured to automatically migrate JMS service to any server instance in the migratable target list. migratable JMS services can also be manually migrated if the host server fails. JMS services can also be manually migrated before performing scheduled server maintenance.

Migratable JMS-related services include:

- JMS Server – a management container for the queues and topics in JMS modules that are targeted to them. See "JMS Server Configuration" on page 3-5.

- Store-and-Forward (SAF) Service – store-and-forward messages between local sending and remote receiving endpoints, even when the remote endpoint is not available at the moment the messages are sent. Only sending SAF agents configured for JMS SAF (sending capability only) are migratable. See Understanding the Store-and-Forward Service in *Configuring and Managing WebLogic Store-and-Forward*.
- Path Service – a persistent map that can be used to store the mapping of a group of messages in a JMS Message Unit-of-Order to a messaging resource in a cluster. One path service is configured per cluster. See "Using the WebLogic Path Service" on page 4-8.
- Custom Persistent Store – a user-defined, disk-based file store or JDBC-accessible database for storing subsystem data, such as persistent JMS messages or store-and-forward messages. See Using the WebLogic Persistent Store in *Configuring Server Environments*.

You can configure JMS-related services for high availability by using migratable targets. A migratable target is a special target that can migrate from one server in a cluster to another. As such, a migratable target provides a way to group migratable services that should move together. When the migratable target is migrated, all services hosted by that target are migrated.

See Understanding the Service Migration Framework in *Using Clusters*.

Automatic Migration of JMS Services

An administrator can configure migratable targets so that hosted JMS services are automatically migrated from the current unhealthy hosting server to a healthy active server with the help of the Health Monitoring subsystem. For more information about configuring automatic migration of JMS-related services, see Roadmap for Configuring Automatic Migration of JMS-Related Services in *Using Clusters*.

Manual Migration JMS Services

An administrator can manually migrate JMS-related services to a healthy server if the host server fails or before performing server maintenance. For more information about configuring manual migration of JMS-related services, see Roadmap for Configuring Manual Migration of JMS-Related Services in *Using Clusters*.

Persistent Store High Availability

As discussed in "What About Failover?" on page 4-5, a JMS service, including a custom persistent store, can be migrated as part of the "whole server" migration feature, or as part of a "service-level" migration for migratable JMS-related services. Migratable JMS-related services

cannot use the default persistent file store, so you must configure a custom file store or JDBC store and target it to the same migratable target as the JMS server or SAF agent associated with the store. (As a best practice, a path service should use its own custom store and migratable target).

Migratable custom file stores can be configured on a shared disk that is available to the migratable target servers in the cluster or can be migrated to a backup server target by using pre/post-migration scripts. For more information on migrating persistent stores, see Custom Store Availability for JMS Services in *Configuring 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 JMS Message Unit-of-Order to a messaging resource in a cluster. It provides a way to enforce ordering by pinning messages to a member of a cluster that is hosting servlets, distributed queue members, or Store-and-Forward agents. One path service is configured per cluster. 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 in a cluster, see Configure path services in the *Administration Console Online Help*.

Path Service High Availability

For high availability, a cluster's path service can be targeted to a migratable target for automatic or manual service migration. However, a migratable path service cannot use the default store, so a custom store must be configured and targeted to the same migratable target. As an additional best practice, the path service and its custom store should be the only users of that migratable target. See Understanding the Service Migration Framework in *Using Clusters*.

Implementing Message UOO With a Path Service

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 4-7.
- 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.

Configuring Foreign Server Resources to Access Third-Party JMS Providers

WebLogic JMS enables you to reference third-party JMS providers within a local WebLogic Server JNDI tree. With Foreign Server resources in JMS modules, 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. Foreign Server resources 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 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

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 the 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 Server Resources

A *Foreign Server* resource in a JMS module 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*.

Note: For information on configuring and deploying JMS application modules in an enterprise application, see Chapter 5, "Configuring JMS Application Modules for Deployment."

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 Factory Resources

A *Foreign Connection Factory* resource in a JMS module 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 Resources

A *Foreign Destination* resource in a JMS module 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 Destination Resources

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

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

Uniform Distributed Destinations vs. Weighted Distributed Destinations

WebLogic Server 9.0 and later offers two types of distributed destination: uniform and weighted. In releases prior to WebLogic Server 9.0, WebLogic Administrators 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 officially referred to as a *weighted distributed destination* (or WDD).

A *uniform distributed destination* (UDD) greatly simplifies the management and development of distributed destination applications.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, Oracle 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 JMS 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.

Note: For information on configuring and deploying JMS application modules in an enterprise application, see Chapter 5, "Configuring JMS Application Modules for Deployment."

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`

Targeting Uniform Distributed Queues and Topics

Unlike standalone queue and topics resources in a module, which can only be targeted 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/91">
  <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 7-2.

Creating Weighted Distributed Destinations

The WebLogic Server Administration Console enables you to configure, modify, target, and delete WDD resources in JMS system modules. 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 in a system module
- Create weighted distributed topics in a system module

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 4-17. 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 4-17.

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 4-17, 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 connection factory 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 4-21.

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 connection factory 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 4-21.

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 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 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 "Migration of JMS-related Services" on page 4-6.

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 Server Failure in *Programming WebLogic JMS*.

CHAPTER 5

Configuring JMS Application Modules for Deployment

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

Methods for Configuring JMS Application Modules

All JMS resources that can be configured in a JMS system module can also be configured and managed as deployable application modules, similar to standard Java EE 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 Java EE modules, configuration changes for an application module are stored in a deployment plan for the module, leaving the original module untouched.

Application developers can use these tools to create and deploy (target) system resources

- Create a JMS system module, as described in "JMS System Module Configuration" on page 3-7 and then copy the resulting `XML` file to another directory and rename it, using "`-jms.xml`" as the file suffix.
- 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.

JMS Schema

In support of the modular deployment model for JMS resources in WebLogic Server 9.*x* or higher, Oracle provides a schema for defining WebLogic JMS resources: `weblogic-jms.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-jms.xsd` schema is available online at `http://www.bea.com/ns/weblogic/weblogic-jms/1.0/weblogic-jms.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.

Packaging JMS Application Modules In an Enterprise Application

JMS application modules can be packaged as part of an Enterprise Application Archive (EAR), as a *packaged module*. 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 Application 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.

Packaged JMS Application Module Requirements

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

- Conforms to the `weblogic-jms.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 Java EE application

Main Steps for Creating Packaged JMS Application 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 Java EE application component's descriptor files, as described in "Referencing a Packaged JMS Application Module In Deployment Descriptor Files" on page 5-4.
6. Package all application modules in an EAR, as described in "Packaging an Enterprise Application With a JMS Application Module" on page 5-9.
7. Deploy the EAR, as described in "Deploying a Packaged JMS Application Module" on page 5-9.

Referencing a Packaged JMS Application 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 Java EE 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 Java EE descriptor file, such as EJB (`ejb-jar.xml`) or WebApp (`web.xml`) files

Referencing JMS Application 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 Java EE 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 Java EE Application components.

Referencing JMS Resources In a Java EE 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>
```

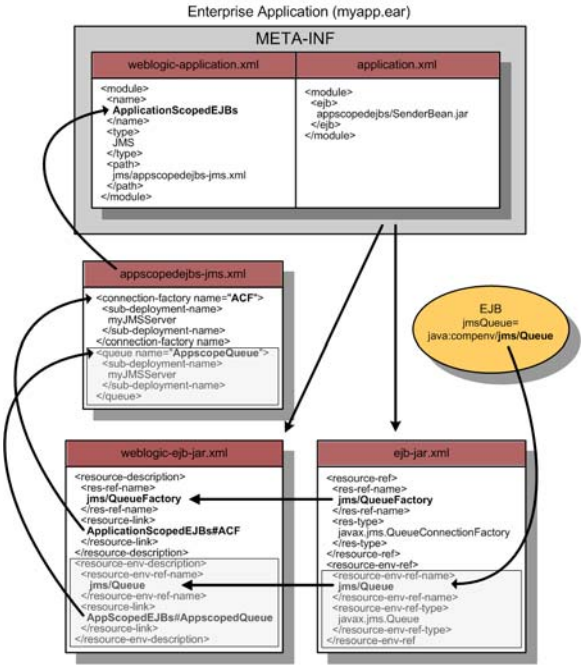
Sample of a Packaged JMS Application Module In an EJB Application

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/91">
  <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 Application Module and Descriptors In an EJB Application



Packaged JMS Application 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 Application 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 Application 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 Application 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 Application 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 Java EE 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 Application Modules

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 modules 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 Java EE applications and modules to WebLogic Server. For information about WebLogic Server deployment, see Understanding WebLogic Server Deployment in *Deploying Applications to WebLogic Server*.

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 Application 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.

Standalone JMS Application Module Requirements

A standalone JMS module must meet the following criteria:

- Conforms to the `weblogic-jms.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 Application 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 Application Modules" on page 5-12.

Sample of a Simple Standalone JMS Application Module

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

```
<weblogic-jms xmlns="http://www.bea.com/ns/weblogic/91">
  <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 Application 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, JMS, and WLDF Application Modules in *Deploying Applications to WebLogic Server*.

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 Application 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 Java EE applications and modules to WebLogic Server. For information about WebLogic Server deployment, see Understanding WebLogic Server Deployment in *Deploying Applications to WebLogic Server*.

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

Generating Unique Runtime JNDI Names for JMS Resources

JMS resources, such as connection factories and destinations, are configured with a JNDI name. The runtime implementations of these resources are then bound into JNDI using the given names. In some cases, it is impossible or inconvenient to provide a static JNDI name for these resources.

An example of such a situation is when JMS resources are defined in a JMS module within an application library. In this case, the library can be referenced from multiple applications, each of which receive a copy of the application library (and the JMS module it contains) when they are deployed. If you were to use static JNDI names for the JMS resources in this case, all applications that refer to the library would attempt to bind the same set of JNDI resources at the same static JNDI name.

Therefore, the first application to deploy would successfully bind the JMS resources into JNDI, but subsequent application deployments would fail with exceptions indicating that the JNDI names are already bound.

To avoid this problem, WebLogic Server provides a facility to dynamically generate a JNDI name for the following types of JMS resources:

- Connection factory
- Destination (queue and topic)
- Weighted distributed destination
- Weighted distributed destination members
- Uniform distributed destination

The facility to generate unique names is based on placing a special character sequence called ${APPNAME} in the JNDI name of the above mentioned JMS resources. If you include ${APPNAME} in the JNDI name element of a JMS resource (either in the JMS module

descriptor, or the `weblogic-ejb-jar.xml` descriptor), the actual JNDI name used at runtime will have the ${APPNAME} string replaced with the effective application ID (name and possibly version) of the application hosting the JMS resource.

Note: The ${APPNAME} facility does not imply that you can define your own variables and substitute their values into the JNDI name at runtime. The string `${APPNAME}` is treated specially by the JMS implementation, and no other strings of the form `${<some name>}` have any special meaning.

Unique Runtime JNDI Name for Local Applications

In the case of JMS modules in a local application, at runtime ${APPNAME} becomes the name/ID of the application. For example:

```
<jndi-name>${APPNAME}/jms/MyConnectionFactory</jndi-name>
```

When deployed within an application called *`MyApp`*, it would result in a runtime JNDI name of:

```
MyApp/jms/MyConnectionFactory
```

Unique Runtime JNDI Name for Application Libraries

In the case of JMS modules in an application library, at runtime ${APPNAME} becomes the name/ID of the application which refers to the library (not the name of the library). For example:

```
<jndi-name>${APPNAME}/jms/MyConnectionFactory</jndi-name>
```

When deployed within an application library called *`MyAppLib`*, and referenced from an application called *`MyApp`*, it would result in a runtime JNDI name of:

```
MyApp/jms/MyConnectionFactory
```

Unique Runtime JNDI Name for Standalone JMS Modules

In the case of JMS modules deployed as stand-alone modules, at runtime ${APPNAME} becomes the name/ID of the stand-alone module. For example:

```
<jndi-name>${APPNAME}/jms/MyConnectionFactory</jndi-name>
```

When deployed within a stand-alone JMS module *`MyJMSModule`*, it would result in a runtime JNDI name of:

```
MyJMSModule/jms/MyConnectionFactory
```

Where to Use the ${APPNAME} String

The ${APPNAME} string can be used anywhere you refer to the JNDI name of a JMS resource. For example, in the:

- `jndi-name` or `local-jndi-name` element of `connection-factory` elements in the JMS module descriptor.
- `jndi-name` or `local-jndi-name` element of `queue` or `topic` elements in the JMS module descriptor.
- `jndi-name` element of `distributed-queue` or `distributed-topic` elements in the JMS module descriptor.
- `jndi-name` element of `uniform-distributed-queue` or `uniform-distributed-topic` elements in the JMS module descriptor.
- `destination-jndi-name` element of `message-destination-descriptor` elements in the `weblogic-ejb-jar.xml` descriptor.

  Note: WebLogic EJB also supports the use of the ${APPNAME} string.
- `jndi-name` element of `weblogic-enterprise-bean` elements in the `weblogic-ejb-jar.xml` descriptor.

Example Use-Case

In a single-server environment, Weblogic Integration Worklist uses application-scoped JMS resources (e.g., queues and connection factories) to support its modular deployment goals. Application-scoped JMS allows Weblogic Integration to have an application library define the EJBs, JMS resources, etc., needed by Worklist, and then have users simply include Worklist into their application by adding a `library-ref` to their application. However, this prevents Worklist user from scaling those destinations to the cluster from an application library.

In a clustered environment, users can now substitute the ${APPNAME} string for the queue's JNDI name at runtime to make the global JNDI names for the queues unique. This way, the JMS ${APPNAME} parameter is replaced at runtime with the application name of the host application being merged to the application library.

CHAPTER 6

Using WLST to Manage JMS Servers and JMS System Module 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 module resources. See Using the WebLogic Scripting Tool and WLST Sample Scripts in *WebLogic Scripting Tool*.

Understanding JMS System Modules and Subdeployments

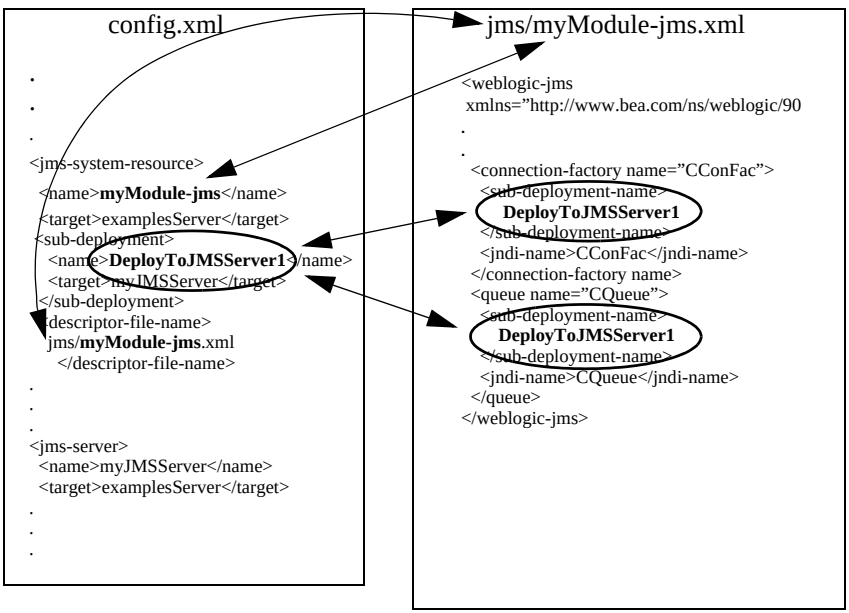
A JMS system module is described by the `jms-system-resource` MBean in the `config.xml` file. Basic components of a `jms-system-resource` MBean are:

- `name`—Name of the module.
- `target`—Server, cluster, or migratable target the module is targeted to.
- `sub-deployment`—A mechanism by which JMS system 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 system 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 system 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 JMS System Module and Resource Subdeployment Targeting 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



How to Create JMS Servers and JMS System Module Resources

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

- Start an edit session.
- Create a JMS system module that includes JMS 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 module 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 JMS 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 System 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 System Module 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 MBeans (WLST Online) 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 Resources

This section provides best practices information when using WLST to configure JMS servers and JMS system module 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.

CHAPTER 7

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:

- "Managing JMS Messages" on page 7-5

For more information about the WebLogic Diagnostic Service, see *Configuring and Using the WebLogic Diagnostics Framework*.

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 resources: JMS servers, connections, queue and topic destinations, JMS server session pools, pooled connections, active sessions, message producers, message consumers, and durable subscriptions on JMS topics.

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 Monitor 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 Server: 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 Server: Monitoring: Active Transactions in the *Administration Console Online Help*.

Monitoring Active JMS Connections, Sessions, Consumers, and Producers

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 Server: Monitoring: Active Connections in the *Administration Console Online Help*.

Using the JMS server's Active Connections monitoring page, you can also monitor statistics on all the active JMS sessions, consumers, and producers on your server. A session defines a serial order for both the messages produced and the messages consumed, and can create multiple message producers and message consumers. The same thread can be used for producing and consuming messages.

For more information on using the Administration Console to monitor session, consumers, and producers, see the following topics in the *Administration Console Online Help*:

- JMS Servers Monitoring: Active Connections: Sessions
- JMS Server: Monitoring: Active Connections: Sessions: Consumers
- JMS Server: Monitoring: Active Connections: Sessions: Producers

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 Server: 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 synchronous 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 Monitor 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 7-5.

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 Durable Subscribers for Topics

You can monitor statistics on all the durable subscribers that are running on your JMS topics via the Administration Console or through the JMSDurableSubscriberRuntimeMBean. Durable subscribers allow you to assign a name to a topic subscriber and associate it with a user or application. WebLogic stores durable subscribers in a persistent file-base store or JDBC-accessible database until the message has been delivered to the subscribers or has expired, even if those subscribers are not active at the time that the message is delivered.

You can use the Administration Console to manage durable subscribers running on topics, as described in "Managing JMS Messages" on page 7-5.

Monitoring Uniform Distributed Queues

You can monitor statistics on uniform distributed queue resources in JMS modules via the Administration Console or through the JMSDestinationRuntimeMBean. 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 Uniform distributed queues - monitor statistics in the *Administration Console Online Help*.

You can also use the Administration Console to manage messages on distributed queues, as described in "Managing JMS Messages" on page 7-5.

Monitoring Uniform Distributed Topics

You can monitor statistics on uniform distributed topic resources in JMS modules via the Administration Console or through the JMSDestinationRuntimeMBean. 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 Uniform distributed queues - monitor statistics in the *Administration Console Online Help*.

Monitoring Pooled JMS Connections

You can monitor statistics on all the active pooled JMS connections on your server. A pooled JMS connection is a session pool used by EJBs and servlets that use a resource-reference element in their EJB or servlet deployment descriptor to define their JMS connection factories.

For more information, see JMS Server: Monitoring: Active Pooled Connections 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 Using Java APIs

WebLogic Java Management Extensions (JMX) enables you to access the JMSDestinationRuntimeMBean and JMSDurableSubscriberRuntimeMBean to manage messages on JMS queues and topic durable subscribers. For more information, see Accessing WebLogic Server MBeans with JMX in *Developing Custom Management Utilities with JMX*.

JMS Message Management Using the Administration Console

The JMS Message Management page of the Administration Console summarizes the messages that are available on the standalone queue, distributed queue, or durable topic subscriber you that 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 (delete) all of the messages from the queue or durable subscription.

For more information on using the Administration Console to manage messages on standalone queues, distributed queues, and durable subscribers, see the following instructions in the *Administration Console Online Help*:

- Manage queue messages
- Manage distributed queue messages
- Manage topic durable subscribers

Each message management function is described in detail in the following sections.

Monitoring Message Runtime Information

By default, the JMS Message Management page displays the information about each message on a queue or durable subscriber 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 forward 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 8-17.

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 or topic durable subscriber 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 8-17.

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 8-17.

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.
- Manage 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 Server: 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*.

Managing Durable Topic Subscribers

You can view a list of durable subscribers for a given topic, browse messages associated with a subscriber, create and delete subscribers, and delete selected messages or purge all messages for a subscription.

For more information, see Manage topic durable subscribers in the *Administration Console Online Help*.

CHAPTER 8

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

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 the 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 Using the Change Center in *Introduction to Oracle WebLogic Server*).
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 Using the Change Center in *Introduction to Oracle WebLogic Server*).

   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 some messages at boot time regarding what store the JMS server is using and its configured destinations.
- DebugJMSDispatcher (scope weblogic.jms.dispatcher) – prints information related to PeerGone() occurrences.
- DebugJMSDistTopic (scope weblogic.jms.config) – prints information about distributed topics, and primary bind and unbind information.
- DebugJMSPauseResume (scope weblogic.jms.pauseresume) – prints information about (backend) pause/resume destination operations.
- DebugJMSModule (scope weblogic.jms.module) – prints a lot of information about JMS module operations and message 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 (store-and-forward) destinations.
- 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.

Messaging Kernel and Path Service Debugging Scopes

You can enable the following registered debugging scopes can be enabled for the messaging kernel and the Path service.

- DebugMessagingKernel (scope weblogic.messaging.kernel) – prints information about the messaging kernal.
- DebugMessagingKernelBoot (scope weblogic.messaging.kernelboot) – prints information about booting the messaging kernal (processing messages).
- DebugPathSvc (scope weblogic.messaging.pathsvc) – prints limited information about some unusual conditions in the path service.

– `DebugPathSvcVerbose` (scope `weblogic.messaging.pathsvcverbose`) – prints limited information about unusual conditions in the path service.

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 in the *Administration Console Online Help*:

- View and configure logs
- Configure JMS server message log rotation
- Configure topic message logging
- Configure queue message logging
- Configure JMS template message logging
- Uniform distributed topics - configure message logging
- Uniform distributed queues - configure message logging

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.messages.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\wlserver_10.0`.

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 in the *Administration Console Online Help*:

- Configure topic message logging
- Configure queue message logging
- Configure JMS template message logging
- Uniform distributed topics - configure message logging
- Uniform distributed queues - configure message logging

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 Module 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 Developing a Basic JMS Application 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 "<" and in place of the right-pointing angle bracket you will see ">" 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"><mes:Int>1</mes:Int></mes:prope
rty></mes:Properties></mes:Header><mes:Body><mes:T
ext/></mes:Body></mes:WLJMSMessage>> <>

Message Expired Event

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

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

Retry Exceeded Event

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

<mes:WLJMSMessage
xmlns:mes="http://www.bea.com/WLS/JMS/Message"><mes:Header><me
s:JMSCorrelationID>bar</mes:JMSCorrelationID><mes:JMSDeliveryM
ode>PERSISTENT</mes:JMSDeliveryMode><mes:JMSExpiration>0&lt
;/mes:JMSExpiration><mes:JMSPriority>4</mes:JMSPriority>&lt
;mes:JMSRedelivered>true</mes:JMSRedelivered><mes:JMSTimestamp
>1116014813453</mes:JMSTimestamp><mes:JMSType>RetryRecord&l
t;/mes:JMSType><mes:Properties><mes:property
name="JMS_BEA_RedeliveryLimit"><mes:Int>1</mes:Int></mes
:property><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 in the *Administration Console Online Help*:

- Pause JMS server message operations on restart
- Pause topic message operations on server restart
- Pause queue message operations on server restart
- Pause JMS template message operations on server restart
- Uniform distributed topics - pause message operations on server restart
- Uniform distributed queues - pause message operations on server restart

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 in the *Administration Console Online Help*:

- Pause JMS server message operations at runtime
- Pause topic message operations at runtime
- Pause queue message operations at runtime

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 in the *Administration Console Online Help*:

- Pause JMS server message operations on restart
- Pause topic message operations on server restart
- Pause queue message operations on server restart
- Pause JMS template message operations on server restart
- Uniform distributed topics - pause message operations on server restart
- Uniform distributed queues - pause message operations on server restart

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 in the *Administration Console Online Help*:

- Pause JMS server message operations at runtime
- Pause topic message operations at runtime
- Pause queue message operations at runtime

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 in the *Administration Console Online Help*:

- Pause JMS server message operations on restart
- Pause topic message operations on server restart
- Pause queue message operations on server restart
- Pause JMS template message operations on server restart
- Uniform distributed topics - pause message operations on server restart
- Uniform distributed queues - pause message operations on server restart

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 in the *Administration Console Online Help*:

- Pause JMS server message operations at runtime
- Pause topic message operations at runtime
- Pause queue message operations at runtime

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 Configuring Suspension of Message Delivery During JMS Resource Outages 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

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.